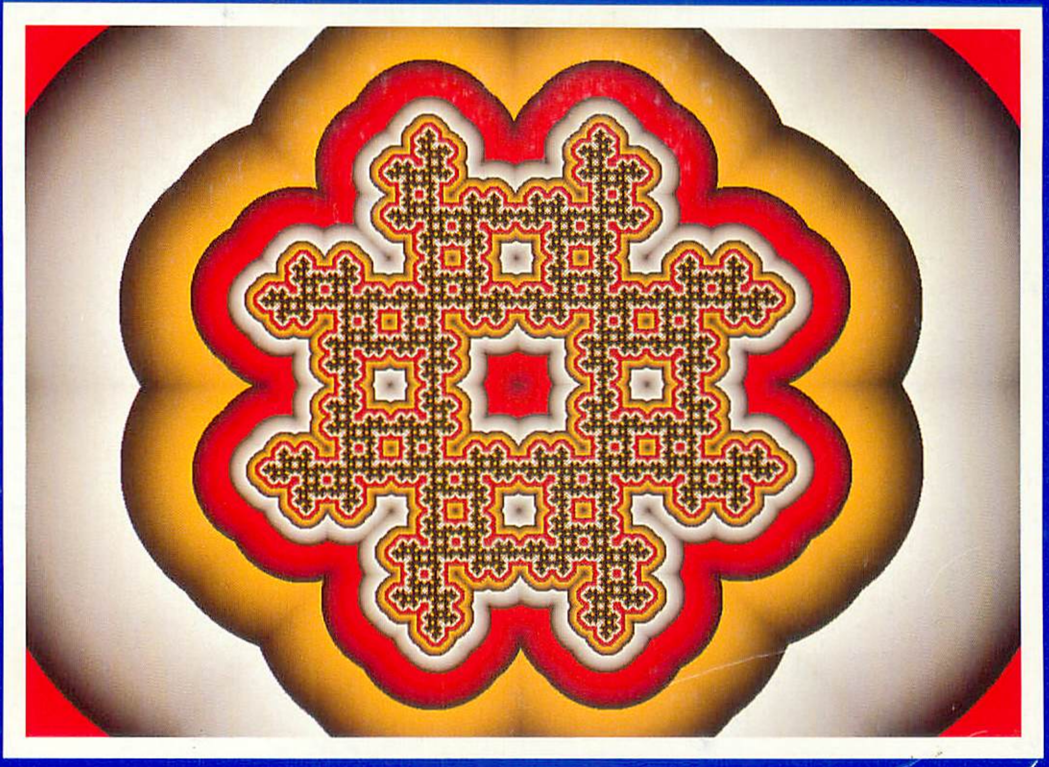
Proceedings

Graphics Interface '92

Compte rendu



11 – 15 May/mai 1992
Vancouver
British Columbia/Colombie-Britannique

Canadian Human-Computer Communications Society

Société canadienne du dialogue humain-machine

ISSN 0713–5424
ISBN 0–9695338–1–0

Conference sponsored by Canadian Human—Computer Communications Society (CHCCS); in cooperation with the Association for Computing Machinery (ACM), the Natural Sciences and Engineering Research Council of Canada and the National Research Council of Canada.

Membership information for the CHCCS is available from:

Canadian Information Processing Society (CIPS)
430 King Street West, Suite 205
Toronto, Ontario, Canada
M5V 1L5
Telephone: (416) 593–4040
Fax: (416) 593–5184

Additional copies of these proceedings are available from:

In Canada
  Canadian Information Processing Society (CIPS)
  (address above)
In the United States
  Morgan Kaufmann Publishers
  Order Fulfillment Center
  P.O. Box 50490
  Palo Alto, CA 94303
  U.S.A.
  Telephone: (415) 965–4081
  Fax: (415) 578–0672
All other countries
  Morgan Kaufmann Publishers
  (address above)

Published by Canadian Information Processing Society

Printed in Canada by Hemlock Printers (Burnaby) Ltd.

Front Cover
The escape–time function of a fractal generated using a language–restricted iterated function system. Created at The University of Calgary by Przemyslaw Prusinkiewicz and Mark Hammel, Department of Computer Science, The University of Calgary.

ISSN 0713–5424
ISBN 0–9695338–1–0

Congrès tenu sous les auspices de la Société canadienne du dialogue humain—machine (SCDHM), en coopération avec l'Association for Computing Machinery (ACM), du Conseil de recherches en sciences naturelles et en génie du Canada et du Conseil national de recherches du Canada.

Des renseignements sur la SCDHM sont disponibles à l'adresse suivante:

l'Association canadienne de l'informatique (ACI)
430 rue King ouest, bureau 205
Toronto (Ontario) Canada
M5V 1L5
Téléphone : (416) 593–4040
Télécopieur : (416) 593–5184

Des exemplaires des comptes rendus sont disponibles aux adresses suivantes :

Au Canada
  L'Association canadienne de l'informatique
  (adresse ci–dessus)
Aux États–Unis
  Morgan Kaufmann Publishers
  Order Fulfillment Center
  P.O. Box 50490
  Palo Alto, CA 94303
  U.S.A.
  Téléphone : (415) 965–4081
  Télécopieur : (415) 578–0672
Tous les autres pays
  Morgan Kaufmann Publishers
  (adresse ci–dessus)

Publié par l'Association canadienne de l'informatique

Imprimé au Canada par Hemlock Printers (Burnaby) Ltd.

Couverture
La fonction de temps de libération d'un fractal créée en utilisant un système de fonctions itérées restreint par un langage. Créé à l'université de Calgary par Przemyslaw Prusinkiewicz et Mark Hammel, département d'informatique, Université de Calgary.

Proceedings / Compte rendu

Graphics Interface '92

Vancouver, British Columbia

11–15 May/mai 1992

Message from the Conference General Co–Chairs

This year Graphics Interface is part of the combined AI/GI/VI '92 Conference held at the University of British Columbia in Vancouver, British Columbia. This year is noteworthy for a number of reasons. In addition to being the second time that all three conferences have been held together, this is the first year that the new name of the society is in effect and the first year that the new conference logo is being used on an on–going basis. Last year CMCCS became CHCCS when "Man–Computer Communications" was changed to "Human–Computer Communications" in keeping with current usage. This year the "eyeball" logo that was originally designed for Graphics Interface/Vision Interface '90 was adopted as a permanent conference logo.

We thank all of the people who helped to organize the conferences and invite all of the attendees to enjoy the technical program, special events and the many pleasures of UBC and Vancouver during their visit.

Kellogg Booth and Alain Fournier
AI/GI/VI '92 Co–Chairs

Message des Coprésidents général du congrès

Cette année Graphics Interface fait partie de la conférence AI/GI/VI '92 qui a lieu à l'université de Colombie Britannique à Vancouver. L'événement est à remarquer pour plusieurs raisons. En plus d'être la deuxième fois que les trois conférences ont lieu en même temps, c'est la première année que le nouveau nom de la société parraine est officiel et que le nouveau logo de la conférence devient permanent. L'année dernière la Société canadienne du dialogue homme–machine est devenue la Société canadienne du dialogue humain–machine (nom qui a l'avantage sur la version anglaise de n'entraîner aucun changement de sigle). Le logo du "globe oculaire" qui avait été conçu initialement pour GI/VI '90 a été aussi adopté cette année de façon permanente.

Nous remercions chaleureusement tous les volontaires qui ont participés à l'organisation de la conférence, et nous invitons tous les participants à profiter du programme de communications techniques, des événements spéciaux et des nombreuses attractions du campus et de Vancouver lors de leur visite ici.

Kellogg Booth et Alain Fournier
AI/GI/VI '92 Co–présidents

Message from the Programme Chair

This year's technical programme consists of 34 carefully selected papers. An increase in submissions this year made our job particularly difficult, but the result is an excellent programme. I would like to draw your attention to the fact that, true to the name *Graphics Interface*, this programme contains original work covering a wide area of computer graphics and human–computer interaction. The programme committee was formed with this in mind, and it performed extremely well in the selection of the most desirable papers from among the many good submissions. I am very grateful to them for making my job easier.

To ensure that each paper was reviewed by three referees, many additional referees were recruited with specific research interests that are related to one or more of the submissions. The quality of refereeing was excellent, and I would like to thank these referees for their contributions to our conference. The names of both the members of the programme committee and of the additional referees are listed below.

This year's conference has seen an increase in participation from Europe and the United States, which makes our conference truly international in character. I thank the authors of all papers for their submissions, whether their papers were accepted or not. I encourage all researchers who read this message to consider submitting a paper to *Graphics Interface '93*.

Eugene Fiume, University of Toronto
Graphics Interface '92 Programme Chair

Message du président du programme

Le programme technique de cette année est fait de 34 communications soigneusement choisies. L'augmentation du nombre de communications soumises a rendu notre tâche particulièrement difficile cette année, mais ça résulte en un excellent programme. Je voudrais souligner que fidèle au nom *interface graphique* ce programme présentent des travaux originaux dans une vaste gamme de sujets en infographie et en interface humain–machine. Le comité du programme a été composé en tenant compte de ce fait, et il s'est très bien acquitté de sa tâche en choisissant les communications les meilleures parmi de nombreuses soumissions de qualité. Je leur suis très reconnaissant d'avoir rendu mon travail plus facile.

Pour assurer que chaque soumission soit considerée par trois arbitres experts, de nombreux arbitres supplémentaires dans des aires de recherche spécifiques ont été recrutés. La qualité des revues a été excellente, et je voudrais remercier ces arbitres pour leur contribution à la conférence. Les noms des membres du comité et des arbitres supplémentaires sont donnés ci–dessous.

Cette année la conférence témoigne d'une augmentation de la participation de l'Europe et des États–Unis, ce qui accroît son charactère international. Je remercie tous les auteurs des communications soumises, qu'elles aient été acceptées ou non. J'encourage tous les chercheurs qui lisent ce message à penser à soumettre une communication à *Graphics Interface '93*.

Eugene Fiume, University of Toronto
Président du programme de Graphics Interface '92

Message from the President of CHCCS

It is with great pleasure that I welcome you to Vancouver and Graphics Interface '92. This is the eighteenth in a series of graphics conferences started in 1969 in Ottawa and held across Canada in various years. It is the longest running graphics conference in the world and continues to provide a significant contribution to the field of computer graphics and interactive techniques. The Canadian Human–Computer Communications Society is pleased to be able to sponsor such a conference and is indebted to all those who have contributed time and effort into making it the success that it is.

Again, following the trend established in 1986 (coincidentally also held in Vancouver), Graphics Interface is being held in conjunction with Vision Interface. Holding these two conferences jointly has helped to promote the interchange of ideas and raise the profile of both conferences. This year we are joined by Artificial Intelligence '92, the Canadian Artificial Intelligence Conference.

It is important that the efforts of all those who have helped to organize these conferences and prepare the technical programs and proceedings be acknowledged for their contributions. Space does not permit us to list all of the many people who have helped out. A partial list appears in the proceedings, but there are many more including the student volunteers and others who have played significant roles behind the scenes in putting on this year's conference.

Wayne A. Davis
President, CHCCS

Message du président du SCDHM

C'est avec grand plaisir que je vous souhaite la bienvenue à Vancouver à la Conférence Graphics Interface '92. Il s'agit de la dix–huitième d'une série de conférences sur l'infographie amorcées en 1969 à Ottawa et présentées un peu partout au Canada au cours des dernières années. C'est aussi la plus longue série au monde de conférences sur l'infographie, et elle continue de'apporter une contribution importante à l'infographie et aux techniques interactives. La Société canadienne du dialogue humain–machine est heureuse de pouvoir parrainer une telle conférence et elle est redevable à ceux et celles qui ont contribué temps et efforts pour en faire un succès.

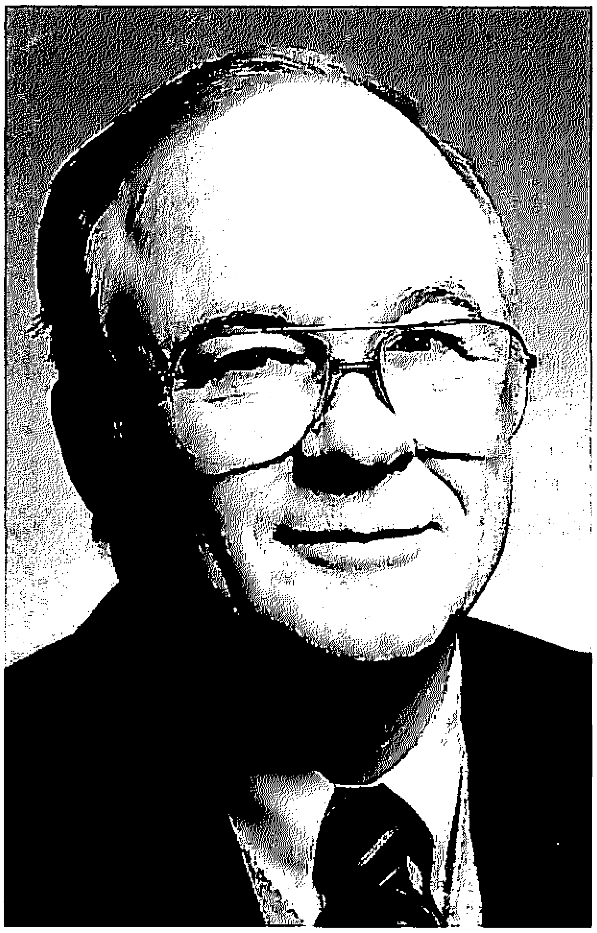
Conformément à la tendance établie en 1986 (par coïncidence aussi à Vancouver), Graphics Interface se tient en même temps que Vision Interface, ce qui contribue à promouvoir l'échange d'idées et à rehausser le profil des deux conférences. Cette année nous sommes aussi avec *Artificial Intelligence '92*, la conférence canadienne sur l'intelligence artificielle.

Enfin, if faut souligner la contribution de tous ceux qui ont aidé à organiser la conférence et à préparer le programme technique et les compte–rendus. Nous sommes très reconnaissants de leurs efforts. Nous n'avons malheureusement pas assez d'espace pour donner une liste complète. Une liste partialle apparaît dans les comptes–rendus, mais il y a de nombreuses personnes, en particuliers les étudiants bénévoles, dont les noms manquent, et qui ont joué un rôle significatif pour que cette conférence ait lieu et soit un succès.

Wayne A. Davis
Président, SCDHM

Organizing Committee/Comité organisateur

Conference '92 General Chairman/Coprésidents général del la conférence de 1992	Kellogg S. Booth, University of British Columbia
	Alain Fournier, University of British Columbia
GI '92 Program Chairman/Président du programme de GI '92	Eugene Fiume, Univerity of Toronto
	John Amanatides, York University
Program Committee/Comité du programme	Tony DeRose, University of Washingtion
	David Forsey, University of British Columbia
	Mark Green, University of Alberta
	Saul Greenberg, University of Calgary
	Przemyslaw Prusinkiewicz, University of Calgary
	Hans–Peter Seidel, University of Waterloo
	Mikio Shinya, Nippon Telephone and Telegraph
	Demetri Terzopoulos, University of Toronto
	Martin Tuori, Alias Research
	Colin Ware, University of New Brunswick
	Marceli Wein, National Research Council/Conseil national de recherches
	Brian Wyvill, University of Calgary
GI '92 Proceedings Editor/Rédacteur du compte rendu	Norman Jaffe
Electronic Theatre/Cinéma électronique	Adele Newton, Alias Research
	Denis Vance, Emily Carr College
Audio Visual/Audio Visuel	Chester Ptasinski, University of British Columbia
Graphic Design/Graphisme	Karin Jager, Jager Design
Local Arrangements/Arrangements locaux	David Poole, University of British Columbia
Registration/Inscription	Lyn Bartram, Simon Fraser University
	Sue Kinderlsey, Simon Fraser University
	Sang Ma, Simon Fraser University
Student Coordinator/Coordinatrice aux étudiants	Scott Flinn, University of British Columbia
	Armin Bruderlin, Simon Fraser University
Treasurer/Trésorier	Fred G. Peet
Administration	Christine Adams, University of British Columbia
Additional Referees/Arbitres supplémentaires	Ronald Baecker, University of Toronto
	Brian Barsky, University of California at Berkeley
	Richard Bartels, University of Waterloo
	John Buchanan, University of British Columbia
	Bill Buxton, University of Toronto
	Tom Carey, University of Guelph
	Laurent Dami, Université de Genève
	George Drettakis, University of Toronto
	David Duce, Rutherford–Appleton Laboratories
	David Ebert, Ohio State University
	Steven Feiner, Columbia University
	Ken Fishkin, Xerox PARC
	Alain Fournier, University of British Columbia
	Sherif Ghali, University of Toronto .
	Simon Gibbs, Université de Genève
	Andrew Glassner, Xerox PARC
	Beverly Harrison, University of Toronto
	Ralph Hill, Bellcore
	Gord Kurtenbach, University of Toronto
	Alison Lee, University of Toronto
	Stephen MacKay, National Research Council/Conseil national de recherches
	Scott MacKenzie, University of Guelph
	Marilyn Mantei, University of Toronto
	Michael McCool, University of Toronto
	Don Mitchell, AT&T Bell Laboratories
	Ken Musgrave, Yale University
	Bruce Naylor, AT&T Bell Laboratories
	Marc Ouellette, University of Toronto
	Michiel van de Panne, University of Toronto
	Pierre Poulin, University of British Columbia
	Kevin Schlueter, University of Toronto
	Gunther Schrack, University of British Columbia
	Thomas Sederberg, Brigham Young University
	Jos Stam, University of Toronto
	James Stewart, University of Toronto
	Mark Tapia, University of Toronto
	Dave Tonnesen, University of Toronto
	Luiz Velho, University of Toronto
	Joe Warren, Rice University
	Andrew Woo, Alias Research
	Charles Wuethrich, University of Toronto



1992 CHCCS Achievement Award — Wayne Davis, University of Alberta

The recipient of this year's award is Wayne Davis. The Society wishes to recognize and acknowledge his unswerving commitment and many contributions to the Society. We also recognize that his many activities have done much to maintain a hospitable climate in Canada for research and development in computer graphics and human–computer interaction.

Wayne was born in Fort Macleod, Alberta in 1931 and received his elementary schooling there. He received a B.S.E. degree in Engineering (Mathematics option) at George Washington University in 1961. Shortly afterwards he went to Ottawa to continue his studies at the University of Ottawa, receiving his MSc and PhD degrees in Electrical Engineering in 1963 and 1967, respectively. Wayne continued his studies while he was a Research Scientist with the Communications Research Centre and its predecessor, the Defence Research Telecommunication Establishment (Ottawa). Since 1969 Wayne has been on the faculty of the University of Alberta and has been a full Professor since 1977. Wayne took an early retirement from the formal side of the University in 1991, but has remained as the first Professor Emeritus in the Department of Computing Science. He was a founding member and later Acting Director of the Alberta Centre for Machine Intelligence and Robotics (ACMIR).

Wayne has wide ranging interests that span computer graphics, image processing and other aspects of computing. His specific research interests have ranged from geographical information systems to image processing and remote sensing. Wayne supervised one of the first PhD students in the Department of Computing Science at the University of Alberta. Subsequently, Wayne supervised many PhD and MSc students, both in image processing and in graphics. His research, and that of his students, has resulted in about one hundred publications in professional journals, conference proceedings and technical reports.

Plaque d'honneur de la SCDHM 1992 — Wayne Davis, Université de l'Alberta

Le lauréat cette année pour recevoir la plaque d'honneur de la SCDHM est Wayne Davis. Nous voulons par là reconnaître son remarquable dévouement et ses nombreuses contributions à la société. Nous voulons aussi reconnaître que ses nombreuses activités sont largement responsables pour avoir établi et maintenu au Canada un climat favorable à la recherche et au developement en infographie et en interaction humain–machine.

Wayne est né à Fort Macleod, Alberta, en 1931, et a reçu là son instruction élémentaire. Il a reçu un baccalauréat en génie (option mathématique) de l'université George Washington en 1961. Peu après il s'est rendu à Ottawa pour poursuivre ses études a l'université d'Ottawa, d'où il a reçu sa maîtrise et son doctorat en 1963 et 1967, respectivement. Wayne a continué ses études tout en étant un chercheur scientifique au Centre de recherche en communications, et à son prédècesseur, l'Établissement de la défense en recherche en télécommunications, à Ottawa. Depuis 1969 Wayne est membre de la faculté de l'université de l'Alberta, et est professeur titulaire depuis 1977. Wayne a pris une retraite anticipée en 1991, mais a guardé contact avec l'université en devenant le premier professeur *emeritus* du département d'informatique. Il a été un membre fondateur, et plus tard directeur *pro tempore* de l'*Alberta Centre for Machine Intelligence and Robotics* (ACMIR).

Le domaine d'intérêts de Wayne recouvre l'infographie, le traitement d'images et beaucoup d'autres aspects de l'informatique. Spécifiquement ses recherches sont allés des sytèmes d'information géographiques au traitement de l'image et à la télé–détection. Wayne a dirigé une des toutes premières thèses de doctorat en informatique à l'université de l'Alberta. Il a ensuite dirigé de nombreux autres étudiants de maîtris et de doctorat, aussi bien en traitement d'image qu'en infographie. Seul ou avec ses étudiants il a publié environ une centaine de communications dans des revues, des conférences ou dans des

In the 1980s Wayne was one of the main participants in the cooperative program with the Department of Computer and Information Science of the Harbin Shipbuilding Engineering Institute, Harbin, People's Republic of China. He has made several visits to Harbin, lecturing there and in other places in the People's Republic of China. Wayne has maintained links with university life in China; since 1985 he has been an Honorary Professor in the Department of Computer Science and Information Science at Harbin Shipbuilding Engineering Institute.

Wayne has been the guiding light and untiring leader of the Canadian Human–Computer Communications Society (CHCCS) since it evolved from its humble beginnings. Through his parallel activities in the Canadian Image Processing and Pattern Recognition Society (CIPPRS) and in the Canadian Society for Computational Studies of Intelligence (CSCSI) he was responsible for bringing the respective conferences under one roof. A first tentative step to such conference was a graphics stream in the CSCSI Conference, organized by Wayne and held in Victoria in 1980 (a CMCCS off–year). After much persuasion by Wayne, either or both of the other two societies joined with CMCCS, and now CHCCS, to hold conferences in succeeding years.

Wayne's interest in, and allegiance to, Western Canada has always been evident in his efforts to make CHCCS national in scope. While the early conferences were held in Ottawa with an occasional foray into other parts of Eastern Canada, Wayne has been instrumental in bringing the conferences to Victoria, Vancouver, Edmonton and Calgary. In each case Wayne used his powers of persuasion to ensure that there was a strong conference organization. Consequently these conferences were highly successful. By making the Society such an ongoing success he has contributed inestimably towards the maintenance of a climate for exchanging research ideas and for providing researchers and especially students with a place to publish and to interact with researchers at other academic institutions.

Through this award, members of the Society wish to acknowledge Wayne's many contributions to the overall development of computer graphics and human–computer interaction research in Canada.

rapports techniques.

Dans les années 1980, Wayne a été un des principaux participants dans le programme coopératif avec le département d'informatique de l'Institut de génie de construction navale d'Harbin, en République de Chine. Il a visité Harbin plusieurs fois, en donnant là et en plusieurs autres endroits de Chine des séminaires. Wayne a maintenu des liens étroits avec le monde universitaire chinois. Depuis 1985 il est professeur honoraire du département d'informatique de l'Institut de génie de construction navale d'Harbin.

Wayne a été le phare et le leader infatigable de la Société canadienne du dialogue humain–machine (SCDHM) depuis ses modestes débuts. A travers ses activités parallèles dans l'Association canadienne de traitement de l'images et de reconaissance des formes (ACTIRF) et de la Société canadienne pour l'étude de l'intelligence par ordinateur (SCEIO) il a été responsable pour l'intégration de leurs conférences sous un seul toit. La première expérience avec une telle conférence a été une filière infographie lors de la conférence de la SCEIO, organisée par Wayne, qui eut lieu à Victoria en 1980 (une année sans conférence SCDHM). Après avoir été convaincues par Wayne, une ou deux des sociétés ont joint SCDHM pour tenir des conférences dans les années suivantes.

L'intérêt et la fidélité de Wayne au Canada de l'ouest a toujours été en évidence, y compris dans ses efforts pour faire de la SCDHM une société à l'échelle nationale. Alors que les premières conférences ont eu lieu à Ottawa, avec des pointes dans d'autres villes de l'est canadien, Wayne a été le moteur principal dans le mouvement des conférences à Victoria, Vancouver, Edmonton et Calgary. Dans chaque cas Wayne a utilisé sa force de persuasion afin de garantir une bonne organisation, et en conséquence ces conférences ont été un grand succès. En assurant ainsi le succès de la société il a contribué immensément à la création et au maintien d'un climat favorable à l'échange des idées et à la présence d'un lieu où les chercheurs et surtout les étudiants chercheurs, peuvent publier et interagir avec leurs collègues des autres sites universitaires.

Par l'attibution de cette plaque, les membres de la société veulent souligner les nombreuses contributions de Wayne au développement général de l'infographie et de l'interaction humain–machine au Canada.

Table of Contents / Table des Matières

An Implementation of Multivariate B-Spline Surfaces over Arbitrary Triangulations

Philip Fong and Hans-Peter Seidel *

Computer Graphics Laboratory,
Department of Computer Science,
University of Waterloo,
Waterloo, Ontario, Canada N2L 3G1

Abstract

Recently in [7], a new multivariate B-spline scheme based on blending functions and control vertices was developed. This surface scheme allows C^{k-1}-continuous piecewise polynomial surfaces of degree k over arbitrary triangulations to be modelled. Actually, piecewise polynomial surfaces over a *refined* triangulation are produced given an arbitrary triangulation. The scheme exhibits both affine invariance and the convex hull property, and the control points can be used to manipulate the shape of the surface locally. This paper describes a test implementation of the scheme for quadratic and cubic surfaces. Issues such as evaluating points on the surface, evaluating derivatives on the surface and representing piecewise polynomial surfaces as linear combinations of B-splines will be discussed. Several examples illustrate the implementation. The work is incorporated into a surface editor which is currently being developed at the University of Waterloo.

Keywords: Blossoming, B-patch, B-spline surface, blending functions, control points, simplex splines, polar forms.

1 Introduction

Tensor-product B-spline surfaces [1, 2, 8, 9, 14, 24] have proven themselves an excellent tool for the modelling of free form surfaces. However, tensor-product surfaces also have their well-known draw-backs if the modelling of largely irregular objects is required. Therefore, not surprisingly, the need for B-splines over non-rectangular regions has been expressed quite early [25].

Splines over arbitrary triangulations of the parameter plane have first been considered in [5, 17]. These multivariate splines are defined as projections of simplices and are therefore called simplex splines. The main drawback of simplex splines in the past has been the difficulty to form linear combinations and the absence of control points.

A different approach has been taken in [30]. The B-patches developed there are based on the study of symmetric recursive evaluation algorithms and are defined by generalizing the de Boor algorithm for the evaluation of a B-spline segment from curves to surfaces. B-patches have control points but the construction of smooth surfaces still requires considerable computation.

Other approaches to the construction of B-splines over irregular domains have been based on subdivision [3, 11], interpolation [21], and on the use of multisided patches [18, 19, 26]. However, each of these schemes has its own difficulties.

One really needs a scheme which constructs automatically smooth complex surfaces and which contains control vertices for shape manipulation. A new multivariate B-spline scheme based on a combination of B-patches and simplex splines which meets these criteria was developed in [7]. This paper discusses details of an implementation of it which is being used in a surface editor being developed at the University of Waterloo. A process of converting piecewise Bézier polynomials to this new scheme and vice-versa will be explained. This leads to a method for surface refinement.

The paper is divided up in the following way. Section 2 introduces some notation which will be used in the remainder of the paper. Section 3 describes the new B-spline scheme. Section 4 discusses the implementation while Section 5 illustrates the new B-spline scheme through examples. We finally finish off with some concluding remarks.

2 Notation and Definitions

This section introduces some notation which is used in the rest of the paper.

Let $W = \{\mathbf{w}_0, \mathbf{w}_1, \mathbf{w}_2\} \subset \mathbb{R}^2$ be a set of affinely independent points and let $\mathbf{u} \in \mathbb{R}^2$. If the determinant

*This work has been partly supported by the Natural Sciences and Engineering Research Council of Canada through Personal Operating Grant OGP0105573 and Strategic Operating Grant STR0040527

$d(W)$ is defined as

$$d(W) = \det \begin{pmatrix} 1 & 1 & 1 \\ \mathbf{w}_0 & \mathbf{w}_1 & \mathbf{w}_2 \end{pmatrix}$$

and the determinant $d_j(\mathbf{u}|W)$ as $d(W)$ with the point $\mathbf{w}_j$ being replaced by $\mathbf{u}$, then the *barycentric coordinates* of $\mathbf{u}$ with respect to the ordered set W are given as

$$\lambda_j(\mathbf{u}) = \frac{d_j(\mathbf{u}|W)}{d(W)}, \quad j = 0\ldots 2 \tag{1}$$

Note that

$$\mathbf{u} = \sum_{j=0}^{2} \lambda_j(\mathbf{u})\mathbf{w}_j \quad \text{and} \quad \sum_{j=0}^{2} \lambda_j(\mathbf{u}) = 1.$$

For a set $V = \{\mathbf{v}_0, \ldots, \mathbf{v}_n\} \subset \mathbb{R}^2$, we let $[V]$ denote the convex hull of V and we let $[V)$ denote the half-open convex hull of V. The definition of the half-open convex hull is given in [29] and is repeated here.

Definition 2.1 (Half-Open Convex Hull) *Given $\mathbf{v}_0, \ldots, \mathbf{v}_n \in \mathbb{R}^2$, the half-open convex hull is then defined as follows: Let ξ be the unit horizontal vector in $\mathbb{R}^2$. A point $\mathbf{u} \in \mathbb{R}^2$ belongs to the half-open convex hull $[\mathbf{v}_0, \ldots, \mathbf{v}_n)$ if and only if there exists a vector η with positive slope and a positive scalar ϵ such that the set $\{s\xi + t\eta \mid 0 < s, t, \quad 0 < s + t < \epsilon\}$ is completely contained in the interior of $[\mathbf{v}_0, \ldots, \mathbf{v}_n]$.*

3 Bivariate B-Splines

The new B-Spline scheme is obtained by *matching* B-patches [28, 30] with simplex splines [5, 17]. By matching we mean that the recurrence relation which describes simplex splines is made to agree with the recurrence relation for B-patches under some conditions. Before we present this, we need some background information on simplex splines and B-patches.

3.1 Simplex Splines

Definition 3.1 (Simplex Splines)
Let $V = \{\mathbf{t}_0, \ldots, \mathbf{t}_m\}$ be a finite set of points in $\mathbb{R}^2$ and let $\mathbf{u}$ be a point in $\mathbb{R}^2$. The simplex spline $M(\mathbf{u}|V) = M(\mathbf{u}|\mathbf{t}_0, \ldots, \mathbf{t}_m)$ is defined recursively as follows. For $V = \{\mathbf{t}_0, \mathbf{t}_1, \mathbf{t}_2\}$ we let

$$M(\mathbf{u}|\mathbf{t}_0, \mathbf{t}_1, \mathbf{t}_2) = \frac{\chi_{[\mathbf{t}_0,\mathbf{t}_1,\mathbf{t}_2)}(\mathbf{u})}{|d(\mathbf{t}_0, \mathbf{t}_1, \mathbf{t}_2)|}, \tag{2}$$

where

$$\chi_{[\mathbf{t}_0,\mathbf{t}_1,\mathbf{t}_2)}(\mathbf{u}) = \begin{cases} 1 & \text{if } \mathbf{u} \in [\mathbf{t}_0, \mathbf{t}_1, \mathbf{t}_2) \\ 0 & \text{otherwise} \end{cases} \tag{3}$$

is the characteristic function on $[\mathbf{t}_0, \mathbf{t}_1, \mathbf{t}_2)$. For $V = \{\mathbf{t}_0, \ldots, \mathbf{t}_m\}$, $m > 2$, we set

$$M(\mathbf{u}|V) = \sum_{j=0}^{2} \frac{d_j(\mathbf{u}|W)}{d(W)} M(\mathbf{u}|V \backslash \{\mathbf{t}_{i_j}\}) \tag{4}$$

where $W = \{\mathbf{t}_{i_0}, \mathbf{t}_{i_1}, \mathbf{t}_{i_2}\}$ is any subset of affinely independent points in V [20]. The points $\mathbf{t}_0, \ldots, \mathbf{t}_m$ are referred to as knots.

It is worth mentioning that the above definition is completely independent of the choice of W [20].

Equations (2) and (4) differ slightly from the ones given in [17, 20] in that we have based the characteristic function on the half-open convex hull $[\mathbf{t}_0, \mathbf{t}_1, \mathbf{t}_2)$ (Definition 2.1) instead of the convex hull $[\mathbf{t}_0, \mathbf{t}_1, \mathbf{t}_2]$. Otherwise, problems arise when the recurrence relation is used for points $\mathbf{u}$ which lie along knot lines (lines connecting any two knots) [20]. The above definition alleviates the problem by modifying the area of support for the B-splines. This is analogous to the case for univariate B-splines [27] where they are non-zero on the half-open interval $[\mathbf{t}_0, \mathbf{t}_1)$ instead of the closed interval $[\mathbf{t}_0, \mathbf{t}_1]$.

The simplex splines $M(\mathbf{u}|V)$ then exhibit the following properties:

- Piecewise polynomial of degree $k = m - 2$
- Local support on the closed convex hull $[V]$
- Non-negative - $M(\mathbf{u}|V) \geq 0$ for all $\mathbf{u} \in \mathbb{R}^2$
- C^{k-1}-continuous everywhere

Further information can be obtained from [5, 6, 15, 16, 17, 20, 32].

Definition 3.1 shows us that plenty of simplex splines exist. The question which remains is how to form linear combinations from them such that piecewise polynomial surfaces over arbitrary triangulations can be constructed. This involves choosing the right simplex splines and the right normalization. These problems may be solved by studying B-patches.

3.2 B-patches

B-patches [30] are a patch representation for polynomial surfaces that arises from generalizing the de Boor algorithm from curves to surfaces [28, 30]. One definition of B-patches is by means of their blending functions $B^I_\beta(\mathbf{u})$.

Definition 3.2 (B-patch Blending Functions)
Let $\Delta(I) = [\mathbf{t}_{i_0}, \mathbf{t}_{i_1}, \mathbf{t}_{i_2}] \in \mathbb{R}^2$, $I = (i_0, i_1, i_2)$ be given along with the additional set of knots $\mathbf{t}_{i_0,0}, \ldots, \mathbf{t}_{i_0,k-1}, \mathbf{t}_{i_1,0}, \ldots, \mathbf{t}_{i_1,k-1}, \mathbf{t}_{i_2,0}, \ldots, \mathbf{t}_{i_2,k-1}$ in $\mathbb{R}^2$ such that $\mathbf{t}_{i_0,0} = \mathbf{t}_{i_0}$, $\mathbf{t}_{i_1,0} = \mathbf{t}_{i_1}$, and $\mathbf{t}_{i_2,0} = \mathbf{t}_{i_2}$. Also, assume that every triple set of knots $(\mathbf{t}_{i_0,\beta_0}, \mathbf{t}_{i_1,\beta_1}, \mathbf{t}_{i_2,\beta_2})$, $0 \leq \beta_0 + \beta_1 + \beta_2 \leq k - 1$ is affinely independent, i.e. $[\mathbf{t}_{i_0,\beta_1}, \mathbf{t}_{i_1,\beta_1}, \mathbf{t}_{i_2,\beta_2}]$ forms a proper triangle. Then, for $\mathbf{u} \in \mathbb{R}^2$, the B-patch blending functions $B^I_\beta(\mathbf{u})$, $|\beta| = k$, of degree k over $\Delta(I)$ are given by the recurrence

$$B^I_{(0,0,0)}(\mathbf{u}) = 1, \tag{5}$$

and

$$B^I_\beta(\mathbf{u}) = \sum_{j=0}^{2} \lambda^I_{\beta-e^j,j}(\mathbf{u}) B^I_{\beta-e^j}(\mathbf{u}), \quad |\beta| > 0. \qquad (6)$$

Terms with negative indices are set to zero and $\lambda^I_{\beta,j}(\mathbf{u}) = d_j(\mathbf{u}|W^I_\beta)/d(W^I_\beta)$ are the barycentric coordinates of $\mathbf{u}$ with respect to $W^I_\beta = \{\mathbf{t}_{i_0,\beta_0}, \mathbf{t}_{i_1,\beta_1}, \mathbf{t}_{i_2,\beta_2}\}$. Here $e^0 = (1,0,0)$, $e^1 = (0,1,0)$, and $e^2 = (0,0,1)$.

The B-patch blending functions form a partition of unity [30], i.e. $\sum_{|\beta|=k} B^I_\beta(\mathbf{u}) = 1$. Every polynomial surface F can be represented as a linear combination of them as follows:

$$F(\mathbf{u}) = \sum_{|\beta|=k} \mathbf{c}^I_\beta B^I_\beta(\mathbf{u}), \quad \mathbf{c}^I_\beta \in \mathbb{R}^3 \qquad (7)$$

where

$$\mathbf{c}^I_\beta = f(\mathbf{t}_{i_0,0}, \ldots, \mathbf{t}_{i_0,\beta_0}, \mathbf{t}_{i_1,0}, \ldots, \mathbf{t}_{i_1,\beta_1}, \mathbf{t}_{i_2,0}, \ldots \mathbf{t}_{i_2,\beta_2}) \qquad (8)$$

are the B-patch control points which form the B-patch control net. Here f represents the *blossom* or *polar form* of F [10, 22, 23]. The representation given by (7) is called the *B-patch*.

The shape of a B-patch is strongly influenced by the shape of its control net. We can form larger surfaces by piecing together individual B-patches. However, the construction of overall smooth surfaces still requires quite a bit of computation. What is needed are blending functions which produce smooth piecewise polynomial surfaces automatically. Simplex splines, which were introduced in Section 3.1, give us just that and by combining them with B-patches, we are led to the new B-spline scheme.

3.3 The New B-Spline Scheme

The development of the B-spline scheme in [7] is based upon the fact that the recurrence relations (2),(4) and (5),(6) agree under the proper renormalization and the proper selection of knots. We now briefly describe its construction.

Let $T = \{\Delta(I) = [\mathbf{t}_{i_0}, \mathbf{t}_{i_1}, \mathbf{t}_{i_2}] \mid I = (i_0, i_1, i_2) \in \mathcal{I} \subseteq \mathbb{Z}^3_+\}$ define a triangulation of $\mathbb{R}^2$ or some bounded domain $D \subset \mathbb{R}^2$. Then, for any two $I, J \in \mathcal{I}$, $\Delta(I) \cap \Delta(J)$ is empty or is a common vertex or edge of $\Delta(I)$ and $\Delta(J)$ (see Fig. 1).

Next, a sequence of knots $\mathbf{t}_{i,0}, \ldots, \mathbf{t}_{i,k}$ is assigned to each vertex $\mathbf{t}_i$ in the triangulation such that $\mathbf{t}_{i,0} = \mathbf{t}_i$ and that any set of three knots is affinely independent. The sequence of knots $\mathbf{t}_{i,0}, \ldots, \mathbf{t}_{i,k}$ is referred to as the cloud of knots associated with the vertex $\mathbf{t}_i$. We are now in a situation to construct simplex splines of degree k over the triangulation T. We consider the following simplex splines:

$$M(\mathbf{u}|V) = M(\mathbf{u}|V^I_\beta) \qquad (9)$$

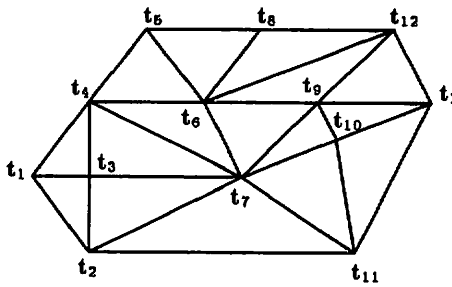


Figure 1: Triangulation of a bounded domain $D \subset \mathbb{R}^2$

where $I \in \mathcal{I}$, $|\beta| = k$, and

$$V^I_\beta = \{\mathbf{t}_{i_0,0}, \ldots, \mathbf{t}_{i_0,\beta_0}, \mathbf{t}_{i_1,0}, \ldots, \mathbf{t}_{i_1,\beta_1}, \mathbf{t}_{i_2,0}, \ldots, \mathbf{t}_{i_2,\beta_2}\}. \qquad (10)$$

We define the regions Ω^I_k as follows:

$$\Omega^I_\beta := \cap_{\gamma\leq\beta}[W^I_\gamma], \quad \Omega^I_k := \mathrm{int}(\cap_{|\beta|=k}\Omega^I_\beta). \qquad (11)$$

We also assume that $\Omega^I_k \neq 0$ which can be obtained if each of the clouds of knots associated with the three vertices of a triangle is kept separate from one another. In other words, for each vertex $\mathbf{t}_i$ in the triangulation T, its cloud of knots is contained within a circle C_i centred at $\mathbf{t}_i$ such that $C_i \cap C_j = 0$, for all $i \neq j$ (i.e., none of the circles intersect one another). Figure 2 below illustrates an example of this setup.

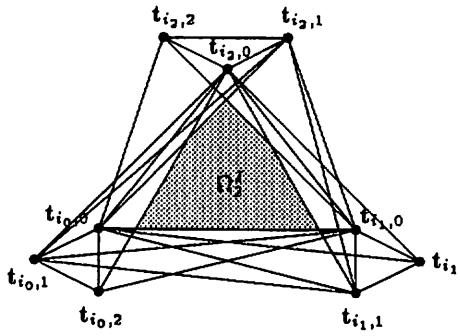


Figure 2: The region Ω^I_2

Then, under these conditions, it is shown in [7] that

$$B^I_\beta(\mathbf{u}) = |d(W^I_\beta)| M(\mathbf{u}|V^I_\beta), \text{ for all } \mathbf{u} \in \Omega^I_k \qquad (12)$$

where $|\beta| = k$, V^I_β is defined in (10) and $W^I_\beta = \{\mathbf{t}_{i_0,\beta_0}, \mathbf{t}_{i_1,\beta_1}, \mathbf{t}_{i_2,\beta_2}\}$. From (12), we let the *normalized B-splines* be defined as

$$N^I_\beta(\mathbf{u}) := |d(W^I_\beta)| M(\mathbf{u}|V^I_\beta). \qquad (13)$$

These will be the blending functions used in the new B-spline scheme. A B-spline surface F of degree k over a given triangulation T with a knot net $\mathcal{K} = \{\mathbf{t}_{i,l} \mid i \in \mathbb{Z}, l = 0, \ldots, k\}$ can then be defined as

$$F(\mathbf{u}) = \sum_{I \in \mathcal{I}} \sum_{|\beta|=k} \mathbf{c}_{I,\beta} N_\beta^I(\mathbf{u}). \tag{14}$$

The $\mathbf{c}_{I,\beta} \in \mathbf{R}^3$ are the control points which make up the control net for the surface F.

Since both simplex splines and B-patches are used to develop the new scheme, their individual properties are inherited by the new scheme. It is these properties which make it possible (relatively easily) to model C^{k-1}-continuous piecewise polynomial surfaces of degree k over arbitrary triangulations.

Affine Invariance: The relationship between the control points and the B-spline surface is invariant under affine coordinate transformations. That is, if $\Phi : \mathbf{R}^3 \to \mathbf{R}^3$ is an affine map (rotation, translation, scaling), then

$$\Phi(\sum_{I \in \mathcal{I}} \sum_{|\beta|=k} \mathbf{c}_{I,\beta} N_\beta^I(\mathbf{u})) = \sum_{I \in \mathcal{I}} \sum_{|\beta|=k} \Phi(\mathbf{c}_{I,\beta}) N_\beta^I(\mathbf{u}). \tag{15}$$

Convex Hull Property: A B-spline surface lies in the convex hull of its control points.

Local Support: Movement of the control point $\mathbf{c}_\beta^I$ only influences the region of the surface on $\triangle(I)$ and those surrounding it.

Continuity A degree k B-spline surface is a piecewise polynomial of degree k over the sub-triangulation induced by its knot net that is C^{k-1}-continuous everywhere if its knots are in general position. But, from the theory of simplex splines, knot multiplicities along a line reduce the order of continuity along this line [20]. For example, a degree 2 surface with knots in general position is C^1-continuous everywhere. Placing three knots on a line reduces the continuity to C^0 and placing four knots produces a discontinuity along the line. Thus, the underlying knot net provides additional degrees of freedom to control the shape of the surface. Figure 3 shows the quadratic normalized B-splines over different knot configurations.

4 Implementation

The theory presented in the previous sections is used in a surface editor which is being developed at the University of Waterloo. A surface editor allows one to manipulate the shape of a surface through the movement of the control vertices which make up the control net. The new B-spline scheme also allows surface changes to be made through movement of knots. This section describes some of the algorithms used in the editor.

4.1 Evaluation

The most important algorithm required is one which evaluates points on the surface. That is, given a parameter value $\mathbf{u} \in \mathbf{R}^2$ in the triangulation T, we want the value of the point on the surface corresponding to $\mathbf{u}$. We use equation (14) as the basic formula in our algorithm. Since the normalized B-splines $N_\beta^I(\mathbf{u})$ are the most complex terms in the equation, we will only concentrate on them.

In order to evaluate the normalized B-splines $N_\beta^I(\mathbf{u})$ defined in (13), we first need to compute the simplex splines $M(\mathbf{u}|V_\beta^I)$ defined by the recurrence (2), (4). We start off by describing the evaluation of linear simplex splines because all higher order splines are composed of these.

Let $V_\beta^I = \{\mathbf{t}_0, \mathbf{t}_1, \mathbf{t}_2, \mathbf{t}_3\} \subset \mathbf{R}^2$ and without loss of generality, let the set $W = \{\mathbf{t}_0, \mathbf{t}_1, \mathbf{t}_2\}$. Then, after expansion of the recurrence and substitution of the base case (2), the linear simplex spline becomes

$$\begin{aligned} M(\mathbf{u}|V_\beta^I) \quad = \quad & \frac{d_0(\mathbf{u}|W)}{d(W)} \frac{\chi_{[\mathbf{t}_1,\mathbf{t}_2,\mathbf{t}_3)}(\mathbf{u})}{|d(\mathbf{t}_1,\mathbf{t}_2,\mathbf{t}_3)|} + \\ & \frac{d_1(\mathbf{u}|W)}{d(W)} \frac{\chi_{[\mathbf{t}_0,\mathbf{t}_2,\mathbf{t}_3)}(\mathbf{u})}{|d(\mathbf{t}_0,\mathbf{t}_2,\mathbf{t}_3)|} + \\ & \frac{d_2(\mathbf{u}|W)}{d(W)} \frac{\chi_{[\mathbf{t}_0,\mathbf{t}_1,\mathbf{t}_3)}(\mathbf{u})}{|d(\mathbf{t}_0,\mathbf{t}_1,\mathbf{t}_3)|}. \end{aligned} \tag{16}$$

One can evaluate (16) by blindly computing and plugging in values for each of the terms. However, depending on the value of the characteristic function χ (3), some of the terms may be zero. This can lead to a very inefficient evaluation technique. A better method is to compute only those terms for which the characteristic function is non-zero. This is, in itself, governed by the choice of W. To do this, we need to know where in the knot configuration the point $\mathbf{u}$ lies. This involves looking at the various knot configurations for a linear spline (see Fig. 4). We first point out that

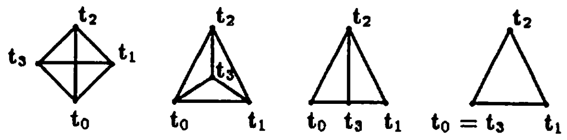


Figure 4: The four essentially different knot configurations for a linear B-spline $M(\mathbf{u}|\mathbf{t}_0, \mathbf{t}_1, \mathbf{t}_2, \mathbf{t}_3)$.

each of the configurations is composed of the four triangles (although some may be degenerate): $\triangle[\mathbf{t}_0, \mathbf{t}_1, \mathbf{t}_2]$, $\triangle[\mathbf{t}_0, \mathbf{t}_1, \mathbf{t}_3]$, $\triangle[\mathbf{t}_0, \mathbf{t}_2, \mathbf{t}_3]$, $\triangle[\mathbf{t}_1, \mathbf{t}_2, \mathbf{t}_3]$. Each of these triangles appears in at most one of the characteristic functions in (16) or in the set W. For every single one of the configurations above, if a point $\mathbf{u}$ is inside its half-open convex hull, then $\mathbf{u}$ belongs to exactly two of its triangles. For instance, in the leftmost configuration above, if $\mathbf{u} \in [\mathbf{t}_0, \mathbf{t}_1, \mathbf{t}_2)$, then either $\mathbf{u} \in [\mathbf{t}_0, \mathbf{t}_1, \mathbf{t}_3)$ or $\mathbf{u} \in [\mathbf{t}_1, \mathbf{t}_2, \mathbf{t}_3)$ but not both. If $\mathbf{u}$ lies on the boundary

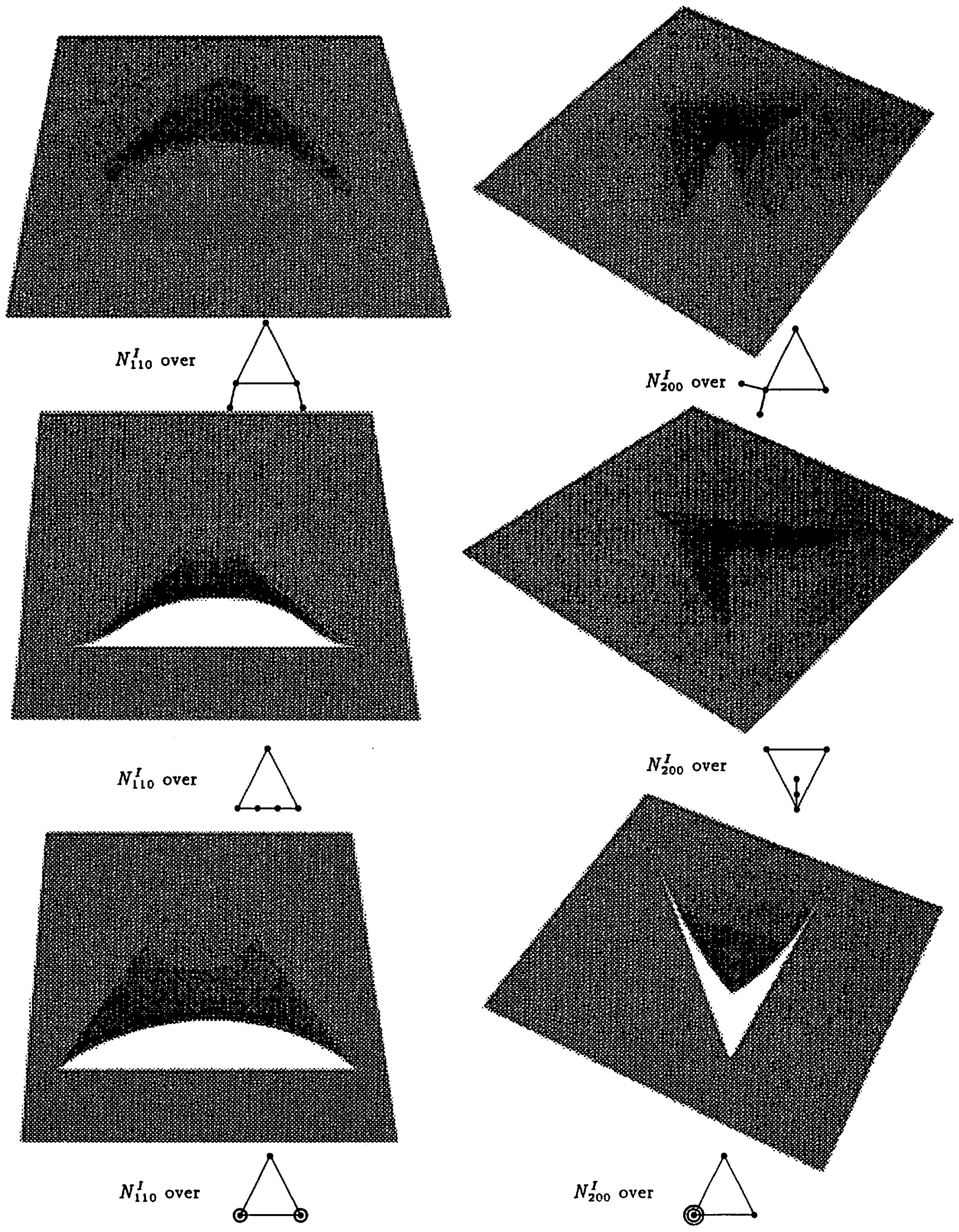


Figure 3: The quadratic normalized B-splines $N^I_{110}(\mathrm{u})$ and $N^I_{200}(\mathrm{u})$ over the six different knot configurations. ⊚ is a double knot and ◎ is a triple knot.

between two triangles or on the vertex of two or more triangles, then Definition 2.1 for the half-open convex hull ensures that $\mathbf{u}$ only belongs to one of them. It is this, which allow points on knot lines to be evaluated correctly [16, 20].

Using these facts then, we let W be one of the triangles in which $\mathbf{u}$ belongs. In light of the above discussion, this will force two of the three characteristic functions to be zero, and hence leaving only one term in (16) to be evaluated.

The only real work involved then is to figure out which triangles the point $\mathbf{u}$ belong. One way is to calculate the barycentric coordinates of $\mathbf{u}$ with respect to the vertices of the triangle. If the coordinates are all greater or equal to zero, then $\mathbf{u}$ is inside, otherwise it's outside. Of course, a slight modification has to be made since we are dealing with the half-open convex hull instead of the closed convex hull.

In the implementation of the surface editor, the computation is organized such that intermediate results obtained from determining the regions containing $\mathbf{u}$ are later re-used in the evaluation of the linear B-spline. In this way, only a small number of determinants need to be explicitly computed while others can be derived from some linear combination of these.

Higher order simplex splines ($m \geq 4$) are simply computed using the recurrence relations (2),(4). We do not try to *optimize* the computation like we did in the linear case above because it is not worthwhile due to the increase in the number of knot configurations. At each level of the recurrence, any choice is suitable for the set $W \subset V^I_\beta$ as long as it forms a proper triangle. However, a good choice is to pick W such that $\mathbf{u} \in [W]$ which gives positive barycentric coordinates. This eliminates any negative terms and hence, increases the numerical stability of the evaluation [16].

Having computed the value for the simplex spline, we can get the value for the normalized B-spline from (13) and finally evaluate the point on the surface from (14).

4.2 Derivatives

A directional derivative along a given direction $\mathbf{v} \in \mathbb{R}^2$ for a parameter value $\mathbf{u} \in \mathbb{R}^2$ may be computed in the same manner as in its evaluation. The only difference lies in the fact that the barycentric coordinates of a vector $\mathbf{v}$ add up to zero instead of one, i.e.

$$\sum_{j=0}^{2} \mu_j(\mathbf{v}) = 0 \text{ and } \mathbf{v} = \sum_{j=0}^{2} \mu_j(\mathbf{v})\mathbf{t}_j. \qquad (17)$$

The directional derivatives for degree k simplex splines is then given as

$$\mathcal{D}_\mathbf{v} M(\mathbf{u}|V) = k \sum_{j=0}^{2} \mu_j(\mathbf{v}) M(\mathbf{u}|V \backslash \{t_{i_j}\}) \qquad (18)$$

with V as defined in Definition 3.1. Then the directional derivative along the direction $\mathbf{v}$ at a parameter value $\mathbf{u}$ for a surface F is given by

$$\mathcal{D}_\mathbf{v} F(\mathbf{u}) = \sum_{I \in \mathcal{I}} \sum_{|\beta|=k} \mathbf{c}^I_\beta \mathcal{D}_\mathbf{v} N^I_\beta(\mathbf{u}) \qquad (19)$$

with

$$\mathcal{D}_\mathbf{v} N^I_\beta(\mathbf{u}) = |d(W^I_\beta)| \mathcal{D}_\mathbf{v} M(\mathbf{u}|V). \qquad (20)$$

We can then use (19) to calculate the tangent normal for a point on the surface in the following way. Any two directional derivatives $\mathbf{v}_1, \mathbf{v}_2 \in \mathbb{R}^2$ ($\mathbf{v}_1 \neq a\mathbf{v}_2, a \in \mathbb{R}$) are computed at the point and the resulting cross-product $\mathcal{D}_{\mathbf{v}_1} F(\mathbf{u}) \times \mathcal{D}_{\mathbf{v}_2} F(\mathbf{u})$ will yield the desired normal vector. The surface editor makes use of these directional derivatives in its surface shading (Gouraud) routines.

4.3 Piecewise Polynomial Surfaces

For a surface scheme to be as flexible as possible, it must be able to represent as many surfaces as possible. This section shows that any piecewise polynomial surface F over a triangulation T can be represented as a linear combination of normalized B-splines $N^I_\beta(\mathbf{u})$. It also shows that B-splines can be represented as piecewise Bézier surfaces.

The precise statement for the representation of piecewise polynomials as linear combinations of B-splines is as follows [31]

Theorem 4.1 *Let F be any piecewise polynomial surface of degree k over a given triangulation T that is C^{k-1}-continuous everywhere and let f_I be the polar form of the restriction of F to the triangle $\triangle(I)$, $I \in \mathcal{I}$. Then*

$$F(\mathbf{u}) = \sum_{I \in \mathcal{I}} \sum_{|\beta|=k} \mathbf{c}_{I,\beta} N^I_\beta(\mathbf{u}) \qquad (21)$$

with

$$\mathbf{c}_{I,\beta} = f_I(\mathbf{t}_{i_0,0}, \ldots, \mathbf{t}_{i_0,\beta_0-1}, \ldots, \mathbf{t}_{i_2,0}, \ldots, \mathbf{t}_{i_2,\beta_2-1}). \qquad (22)$$

If we let $F \equiv 1$, then its polar form $f \equiv 1$ and from Theorem 4.1, we get $\sum_{I,\beta} N^I_\beta(\mathbf{u}) = 1$ which shows that the normalized B-splines $N^I_\beta(\mathbf{u})$ form a global partition of unity.

Piecewise Bézier surfaces F of degree k over irregular triangulations T can be converted to B-spline surfaces of the same degree by using Theorem 4.1. Briefly, the algorithm is as follows. For each vertex $\mathbf{t}_i$ of a given triangulation T of a bounded region $D \subseteq \mathbb{R}^2$, knots $\mathbf{t}_{i,0}, \ldots, \mathbf{t}_{i,k}$ are assigned (in general position) to it such that $\mathbf{t}_{i,0} = \mathbf{t}_i$. The assignment must follow the conditions in Section 3.3 and in addition, vertices on the boundary of D must have their knots outside of D. Then, polar forms f_I of the restriction of F to every $\triangle(I) \in T$, $I \in \mathcal{I}$, are computed using the multiaffine version of the de Casteljau algorithm [10, 23]. The B-spline control points $\mathbf{c}^I_\beta$ are then obtained by evaluating f_I using (22).

Converting degree k B-spline surfaces F over arbitrary triangulations T to piecewise Bézier surfaces $\tilde{F}$ requires a bit more work. We need to come up with two things: a triangulation and the Bézier control points. We cannot use T as our triangulation, as in previous case, because we now need a finer triangulation due to the additional lines introduced by the knots. A finer triangulation can be derived from the knot net associated with each $\triangle(I) \in T, I \in \mathcal{I}$ (see Fig. 2). Let's consider only the $\triangle[\mathbf{t}_{i_0,0}, \mathbf{t}_{i_1,0}, \mathbf{t}_{i_2,0}]$ in Figure 2 and its interior including all line segments passing through the interior We have, in effect, divided up the triangle into regions using the knot lines. Note that not all of the regions in $\triangle[\mathbf{t}_{i_0,0}, \mathbf{t}_{i_1,0}, \mathbf{t}_{i_2,0}]$ are triangular; so, we must further divide (arbitrarily) these regions up. The resulting construction yields the refined triangulation $\tilde{T}_I$ restricted to $\triangle(I)$. Then, the refined triangulation for the piecewise Bézier surface $\tilde{F}$ is $\tilde{T} = \cup_I \tilde{T}_I, \quad \triangle(I) \in T, \quad I \in \mathcal{I}$.

Having constructed the triangulation, we next deal with the Bézier control points. For each domain triangle $\triangle(\tilde{I}) \in \tilde{T}$, we associate with it, a Bézier triangle with control vertices $\mathbf{b}^{\tilde{I}}_{\beta}$. From the theory of Bézier triangles [12], a Bézier surface interpolates the corner vertices (those which lie at the corners of the domain triangles) of its control net. Thus, these *corner* control vertices will precisely lie on the surface F and can be computed by evaluating $F(\tilde{\mathbf{t}}_i)$ for all vertices $\tilde{\mathbf{t}}_i$ in the triangulation $\tilde{T}$. The other control vertices are given by

$$\mathbf{b}^{\tilde{I}}_{\beta} = f(\underbrace{\tilde{\mathbf{t}}_{i_0,0}, \ldots, \tilde{\mathbf{t}}_{i_0,0}}_{\beta_0 \text{ times}}, \ldots, \underbrace{\tilde{\mathbf{t}}_{i_2,0}, \ldots, \tilde{\mathbf{t}}_{i_2,0}}_{\beta_2 \text{ times}}) \tag{23}$$

where f is the multiaffine definition of F.

4.4 Refinement

For practical purposes, surface schemes must also allow for refinement or subdivision [4, 13]. The idea is that fine detail may be required for parts of the surface but the existing control points do not allow for the modelling of such detail. Thus, we need to be able to add *extra* control points only to those regions.

Then, for the new B-spline scheme, we need to have a finer triangulation over the areas that need to be refined. A finer triangulation will, in effect, provide us with more control vertices. We use a combination of the conversion algorithms from Section 4.3 to solve the problem.

Suppose we are given a B-spline surface F over an arbitrary triangulation T and we want to refine the surface region F_I that's restricted to $\triangle(I) \in T$. First, F_I is converted into a piecewise Bézier surface $\tilde{F}_I$ using the latter algorithm in Section 4.3. Then, the resulting Bézier surface $\tilde{F}_I$ can be converted back into B-spline representation using Theorem 4.1. Hence, a finer triangulation with control vertices over $\triangle(I)$ has been produced. This technique also lends itself to recursive refinement (i.e. refined areas may be further refined, etc).

5 Examples

This section illustrates examples of quadratic surfaces produced from our test implementation. The creation of the triangulations and the positioning of the knots for the surfaces were all done manually — no automatic procedure was involved. However, the B-spline editor was used to position the control vertices. The examples show advantages and applications of the new B-spline scheme.

Figure 5 shows the advantage of converting a C^1-continuous piecewise polynomial quadratic Bézier surface into a quadratic B-spline surface. The movement of a Bézier control point will generally destroy the continuity of the surface (Fig. 5(b)), but the movement of a B-spline control point will preserve the smoothness and C^1-continuity throughout the entire surface (Fig. 5(d)). Thus, when designing an object, one does not need to worry about preserving its smoothness, but, can concentrate solely on designing its shape.

Figure 6 shows two examples of an application to the *polygonal hole problem*. This problem involves a degree k piecewise polynomial surface containing an interior hole. We wish to *patch* up the hole such that the overall smoothness or continuity of the surface is preserved (especially around the boundary of the hole). The idea of the solution is the following. We first represent the piecewise polynomial surface around the hole as a linear combination of B-splines (Theorem 4.1). Then, this B-spline surface is extended into the hole to produce an overall C^{k-1}-continuous fill of the hole.

6 Conclusion

The new B-spline scheme offers a method of modelling complex and irregular objects over arbitrary triangulations. Smoothness, locality and the modelling of discontinuities are inherited from simplex splines while control points, affine invariance, and the representation of piecewise polynomials are obtained from B-patches.

The implementation that is presented in this paper has succeeded in demonstrating the practical feasibility of the fundamental algorithms underlying the new surface scheme. Quadratic and cubic surfaces over arbitrary triangulations can be edited and rendered in real-time. Applications like the filling of polygonal holes demonstrate the potential of the new scheme when dealing with concrete design problems. Further improvements to our editor that simplify user input and additional applications are currently under way.

References

[1] R.H. Bartels, J.C. Beatty, and B.A. Barsky. *An Introduction to Splines for Use in Computer Graphics and Geometric Modeling.* Morgan Kaufmann Publishers, 1987.

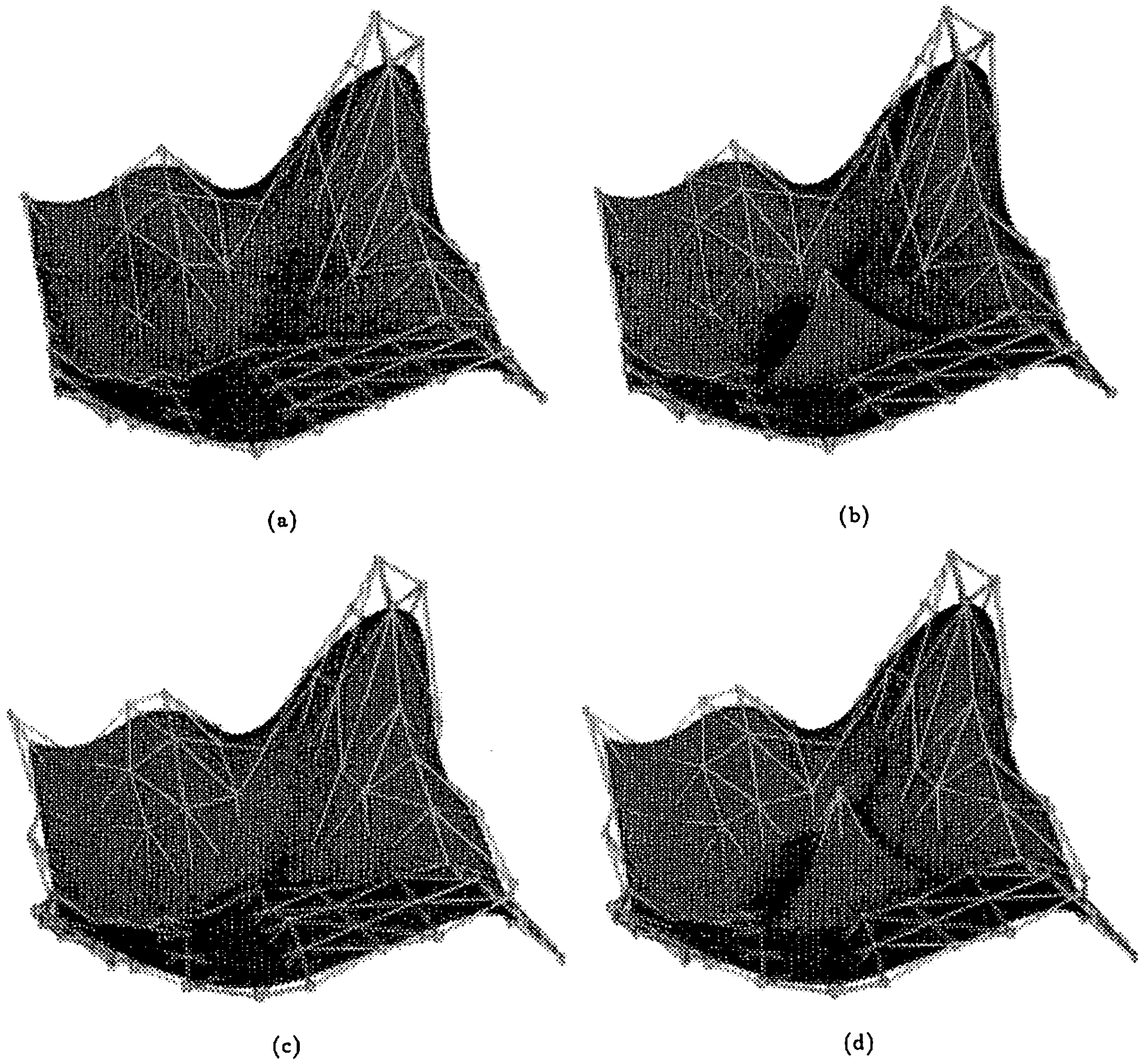
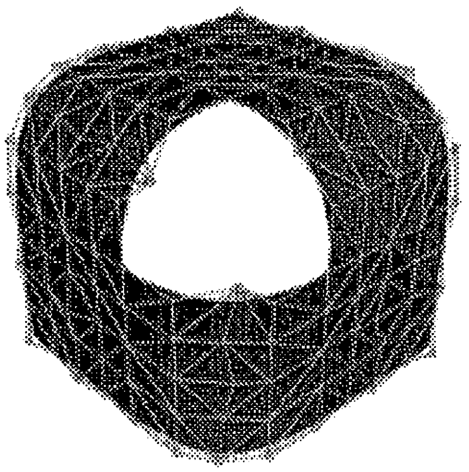
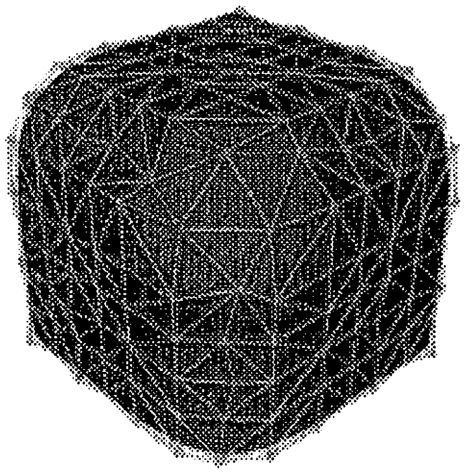
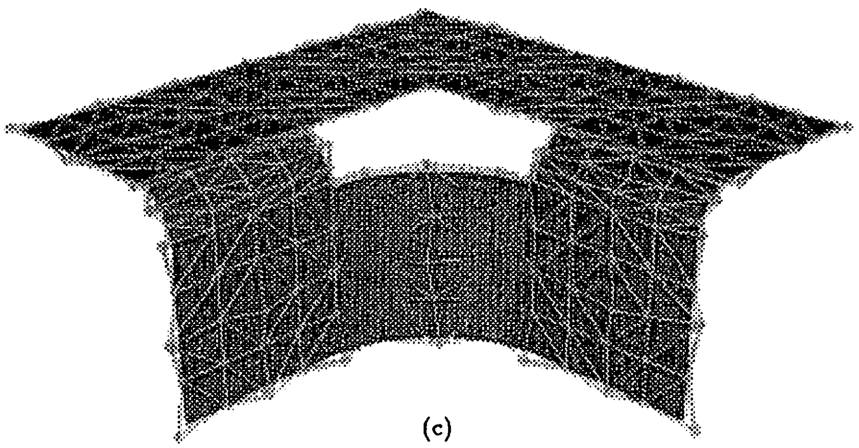


Figure 5: (a) and (b) A quadratic C^1-continuous piecewise polynomial test surface with Bézier control net. (c) and (d) Same surface but with a B-spline control net. Influence of moving a single control point: If a Bézier control point is moved, the C^1-continuity of the surface is destroyed, and a sharp edge is introduced (b). If a B-spline control point is moved, the C^1-continuity of the surface is preserved (d).


(a)


(b)


(c)

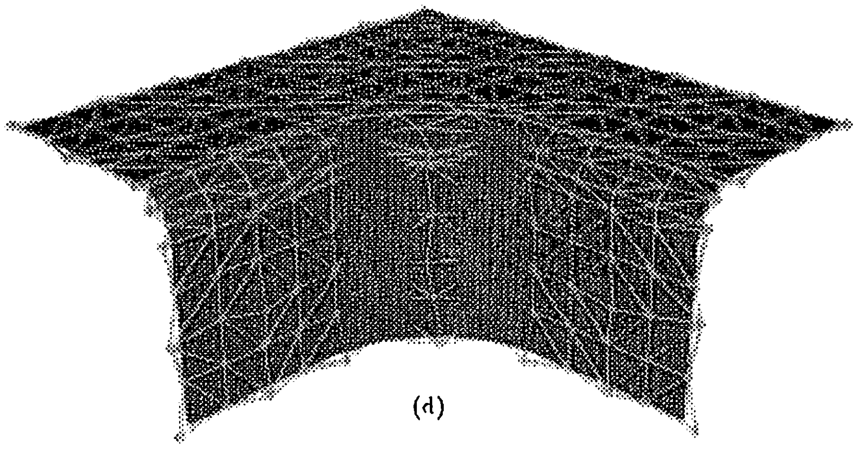

(d)

Figure 6: Solving the polygonal hole problem using triangular B-splines: (a) a C^1-continuous quadratic surface containing a 3-sided interior hole. (b) the hole in (a) is filled and the resulting surface is still overall C^1-continuous. (c) and (d) same problem but with a quadratic surface containing a 5-sided hole.

[2] W. Boehm, G. Farin, and J. Kahmann. A survey of curve and surface methods in CAGD. *Computer-Aided Geom. Design*, 1:1–60, 1984.

[3] E. Catmull and J. Clark. Recursively generated B-spline surfaces on arbitrary topological meshes. *Comput. Aided Design*, 10:350–355, 1978.

[4] E. Cohen, T. Lyche, and R.F. Riesenfeld. Discrete B-splines and subdivision techniques in computer aided geometric design and computer graphics. *Computer Graphics and Image Processing*, 14:87–111, 1980.

[5] W. Dahmen and C.A. Micchelli. On the linear independence of multivariate B-splines I. Triangulations of simploids. *SIAM J. Numer. Anal.*, 19:993–1012, 1982.

[6] W. Dahmen and C.A. Micchelli. Recent progress in multivariate splines. In C.K. Chui, L.L. Schumaker, and J.D. Ward, editors, *Approximation Theory IV*, pages 27–121. Academic Press, 1983.

[7] W. Dahmen, C.A. Micchelli, and H.-P. Seidel. Blossoming begets B-Splines built better by B-Patches. Research Report RC 16261(#72182), IBM Research Division, Yorktown Heights, 1990. Accepted for publication in Math. Comp..

[8] C. de Boor. On calculating with B-splines. *J. Approx. Th.*, 6:50–62, 1972.

[9] C. de Boor. *A Practical Guide to Splines*. Springer, New York, 1978.

[10] P. de Casteljau. *Formes à Pôles*. Hermes, Paris, 1985.

[11] D. Doo and M.A. Sabin. Behaviour of recursive division surfaces near extraordinary points. *Comput. Aided Design*, 10:356–360, 1978.

[12] G.E. Farin. *Curves and Surfaces for Computer Aided Geometric Design*. Academic Press, 1988.

[13] R.N. Goldman. Blossoming and knot insertion algorithms for B-spline curves. *Computer-Aided Geom. Design*, 7:69–81, 1990.

[14] W.J. Gordon and R.F. Riesenfeld. B-spline curves. In R.E. Barnhill and R.F. Riesenfeld, editors, *Computer Aided Geometric Design*. Academic Press, 1974.

[15] T.A. Grandine. The computational costs of simplex spline functions. *SIAM J. Numer. Anal.*, 24:887–890, 1987.

[16] T.A. Grandine. The stable evaluation of multivariate simplex splines. *Math. Comp.*, 50:197–205, 1988.

[17] K. Höllig. Multivariate splines. *SIAM J. Numer. Anal.*, 19:1013–1031, 1982.

[18] C. Loop and T.D. DeRose. A multisided generalization of Bézier surfaces. *ACM Trans. Graph.*, 8(3):204–234, 1989.

[19] C. Loop and T.D. DeRose. Generalized B-spline surfaces of arbitrary topology. In *Proc. SIGGRAPH'90*, pages 347–356. ACM SIGGRAPH, 1990.

[20] C.A. Micchelli. On a numerically efficient method for computing with multivariate B-splines. In W. Schempp and K. Zeller, editors, *Multivariate Approximation Theory*, pages 211–248, Basel, 1979. Birkhäuser.

[21] J. Peters. *Fitting smooth parametric surfaces to 3D data*. PhD thesis, University of Wisconsin, Madison, 1990.

[22] L. Ramshaw. Blossoming: A connect-the-dots approach to splines. Technical report, Digital Systems Research Center, Palo Alto, 1987.

[23] L. Ramshaw. Blossoms are polar forms. *Computer-Aided Geom. Design*, 6:323–358, 1989.

[24] R.F. Riesenfeld. *Applications of B-spline Approximation to Geometric Problems of Computer Aided Design*. PhD thesis, Syracuse University, 1973.

[25] M.A. Sabin. *The Use of Piecewise Forms for the Numerical Representation of Shape*. PhD thesis, Hungarian Academy of Sciences, Budapest, Hungary, 1976.

[26] M.A. Sabin. Non-rectangular surface patches suitable for inclusion in a B-spline surface. In *Proc. EUROGRAPHICS'83*, pages 57–69. North-Holland, 1983.

[27] L.L. Schumaker. *Spline Functions: Basic Theory*. John Wiley & Sons, New York, 1981.

[28] H.-P. Seidel. Algorithms for B-patches. In *Proc. Graphics Interface '91*, pages 8–15. Morgan Kaufmann Publishers, 1991.

[29] H.-P. Seidel. Polar forms and triangular B-Spline surfaces. In *Blossoming: The New Polar-Form Approach to Spline Curves and Surfaces, SIGGRAPH '91 Course Notes #26*, pages 8.1–8.52. ACM SIGGRAPH, 1991.

[30] H.-P. Seidel. Symmetric recursive algorithms for surfaces: B-patches and the de Boor algorithm for polynomials over triangles. *Constr. Approx.*, 7:257–279, 1991.

[31] H.-P. Seidel. Representing piecewise polynomials as linear combinations of multivariate B-splines. In T. Lych and L.L. Schumaker, editors, *Curves and Surfaces*. Academic Press, to appear 1992.

[32] C.R. Traas. Practice of bivariate quadratic simplicial splines. In W. Dahmen, M. Gasca, and C.A. Micchelli, editors, *Computation of Curves and Surfaces*, pages 383–422, Dordrecht, 1990. NATO ASI Series, Kluwer Academic Publishers.

Interactive Solid Geometry Via Partitioning Trees

Bruce F. Naylor

AT&T Bell Laboratories
Murray Hill, NJ 07974
naylor@research.att.com

Abstract

The extension from interactive 2D wireframe geometry to interactive solid geometry has been for some time one of the goals of Computer Graphics. Our approach to this objective is the utilization of a computational representation of geometric sets that we believe is better suited to geometric computation than alternatives inherited from mathematics. This representation is the binary space partitioning tree. Employing such a representation leads to simpler and faster algorithms and has enabled us to construct an elementary interactive solid geometry system which can execute effectively on workstations with no graphics acceleration hardware. Interactive manipulation of a collection of polyhedral objects is provided utilizing set operations, affine transformations, collision detection, picking and dragging, calculation of metric properties, rendering of transparent objects, and solid clipping to an arbitrary polygonal view volume. This is the first such system based on partitioning trees, and as a consequence, is the first interactive system that fully supports set operations, collision detection and transparency for arbitrary polyhedral objects.

Introduction

Real-time manipulation of 3D geometry has since its inception been one the objectives of computer graphics. But only limited success has been made in achieving this goal in the almost three decades that have passed since the creation by Ivan Sutherland of the 2D interactive design program SketchPad. We believe one of the crucial factors for this is the representation. Given any semantic domain, such as geometry, for which we wish to construct a computational object, the selection of the representation largely determines the algorithms required for implementing the operators of the domain. A good representation should yield simple and efficient algorithms, but this will happen only if the representation somehow captures the computational aspects of the semantic domain. While it is possible to invent new data structures optimized for each new problem, this approach, if used in the creation of an integrated software system, is quite impractical. It is also impractical if one is interested in hardware acceleration and/or parallelization. Rather support of only a few important representations is plausible. Such is the case for numbers, where we have only a few integer and floating point representations.

Boundary representations (b-reps), essentially inherited from mathematics, represent polyhedra in terms of topology: i.e. connected components and incidence. No where is the notion of a process, i.e. computation, overtly manifested, and the view is one of an empty space populated by objects. In contrast, the *binary space partitioning tree*, also *bsp tree* or *partitioning tree*, induces a structure on space that reflects the information density of the geometry in that space. It does this via recursive subdivision of space by hyperplanes which results in a hierarchy of convex regions. The combinatorial representation of this is a binary tree, which can also be interpreted as a computation graph or decision tree. Thus computation is intrinsic to the representation. Partitioning trees provide an efficient method of answering spatial relationship questions such as those needed for set operations and visibility determination; in contrast, topological representations have, by abstraction, removed this information from the representation since homeomorphisms do not change its structure.

Partitioning trees originated as a method of pre-processing a polygonal database to facilitate hidden surface removal (see [Schumacker et al 69] and [Fuchs, Kedem and Naylor 80]). Their first use for interactive viewing of solids was described in [Fuchs, Abrams and Grant 83]. In [Thibault and

Naylor 87] partitioning trees were extended to provide representations of polytopes, and algorithms were presented for converting a CSG expression on b-reps into a partitioning tree and for performing a set operation between a partitioning tree and a b-rep by modifying the partitioning tree. This later capability was the basis for an interactive solid modeling program in which the user sculpted a work-piece with a tool; the work-piece was a partitioning tree and the tool was a convex b-rep [Naylor 90]. Solid near-plane clipping was also incorporated. In [Chin and Feiner 89], a partitioning tree based shadow algorithm was presented, and its effectiveness was demonstrated through an interactive program.

The system described in this paper is a successor to [Naylor 90] and represents an entirely new software effort based on work presented in [Naylor, Amanatides and Thibault 90] which provided set operations between arbitrary polytopes via merging of partitioning trees. This new system allows any number of non-convex polyhedra each represented by a partitioning tree. Set operations can be performed between any pair; thus the restriction in [Naylor 90] to a single non-convex work-piece operated on by pre-defined convex tools is removed. In addition, arbitrary affine transformations can be applied to any object and metric properties, such as volume and center of mass, can be calculated.

The system described in this paper is the first interactive system using only partitioning trees for representing polyhedra. Two significant benefits, in addition to set operations, accrue from this: inexpensive collision detection and non-refractive transparency. We believe our system is the first interactive solid geometry system with such capabilities.

User View

The user is presented with a world composed of a set of polyhedral objects, with access to this world through a user-interface built upon a 3-button mouse and pop-up menus. Interactive control of objects and views for models comprised of several thousand faces is possible, even on workstations with no graphics drawing hardware.

The geometry of any object can be modified by applying affine transformations or by performing set operations between it and other objects. Visual attributes, such as color, opacity and texture, can be selected as well. Measurement of any object is also available: this includes computing its volume, mass, surface area, center of mass, moments of inertia, and axis-aligned extents. The object editing operations include choosing a new object from a pre-defined set, copying an object, deleting an object, reading an object from a file, and writing an object to a file. Finally, its combinatorial representation can be displayed (a binary tree).

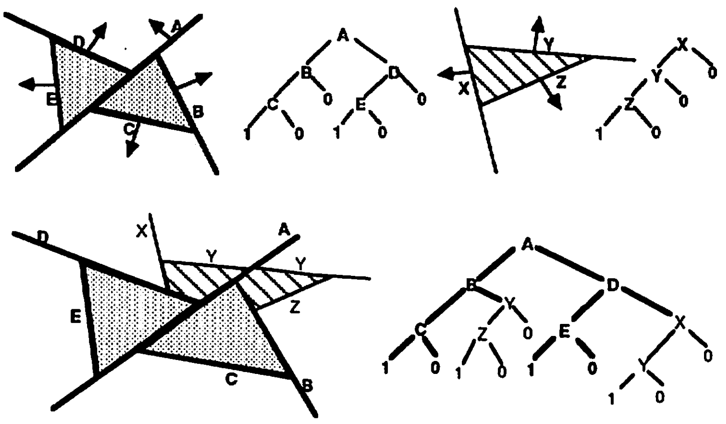
There is at any one time a single "current" object that is the operand of all unary operations on objects, and we refer to this current object as the *tool-object*. The user may at any time select any of the objects as the tool-object via picking (with the mouse). The tool-object is also one of the two operands to any set operation, the other operand being any object in which it is in contact. The tool-object is used as a tool to modify these "in-contact" objects, but the tool-object itself is unaffected. A special sweep mode is provided in which the tool-object is subtracted from each in-contact object as the user moves the tool-object.

Set operations

All polyhedra are represented by partitioning trees, and set operations are performed by merging trees [Naylor, Amanatides and Thibault 90]. Figure 1 gives a simple example suggesting how this could occur. Hyperplane normals point to the positive halfspace which is associated with the right-child of an internal node and the negative halfspace is associated with the left-child. At the leaf-nodes, a 1 indicates an in-cell and a 0 an out-cell. Set operations are achieved by merging two trees. This can be thought of in terms of inserting one tree into the other via a recursive algorithm. At each internal node v, the inserted tree **T** is split by the hyperplane associated with v, and the positive and negative "halves" of **T** are inserted into the respective subtrees of **v**. This continues recursively until a cell is reached. If the operation is, for example, union, then if the cell is an in-cell, the inserted tree is discarded; otherwise, being an out-cell, the cell is replaced with the subset of **T** that has reached/lies-in this cell.

Besides the traditional set operations of union, intersection and difference, we have included a symmetric difference corresponding to the boolean not-equal or exclusive-or. The result of this operation when viewed from the outside looks like union, but where the two objects intersect, a cavity is created which may be seen after subsequent set operations, or temporary cut-aways, or whenever the result is semi-transparent.

As suggested in Figure 1, each in-cell of a partitioning tree has an associated set of *attributes* defining the value of space within that cell (the polka-dots and cross-hatches). Since we are modeling physical objects, we use attributes describing those physical properties of the material being modeled that are relevant to our geometric operations. Thus color, opacity, density, etc., are specified for each in-cell, but not for out-cells. Polyhedra are then viewed as functions from 3-space to an attribute space in which the domain of



A union-operation using BSP Trees
Figure 1

the function is confined to the polyhedra. That is, for polyhedron **P**, point **X**, and attributes **A**, we have the function $\mathbf{f} : \mathbf{X} \in \mathbf{P} \rightarrow \mathbf{A}$, and for $\mathbf{X} \notin \mathbf{P}$, $\mathbf{f}$ is undefined (or maps to the NULL value).

When performing union or intersection, two possibly different attributes are defined wherever the two objects intersect, but the result must specify only one set of attributes in any region of space (i.e. we want a function). We currently have adopted the policy of attribute precedence: the resulting attribute is simply that of the operand with greater precedence, and we always give precedence to the first operand (it seems oddly convenient that the two operations for which this is an issue are commutative). An alternative policy would be to construct a new attribute from the blending of the two operand attributes.

We have also provided support for *transparency*; and by this we mean the thin-film variety, as with colored cellophane, that can be treated as an absorbing but not refracting medium. (This is also to be distinguished from "screen-door" transparency provided by some z-buffer based systems.) The requisite blending calculations are performed during polygon scan-conversion using an "alpha-channel" to hold the opacity of a point on those workstations having such capability. Since the in-cells of a partitioning tree may differ in their material attributes, and since reflection and refraction are considered to occur wherever the electromagnetic properties of space change discontinuously, visible boundaries can be present in the interior of a polyhedron, unlike opaque polyhedra. What is more, such a boundary can be visible from both sides, although with differing filtering properties, if the intervening space is sufficiently transparent. (The boundary between a red region and a green region will appear, if looking from the green region as red, but will appear as green if looking from the red region.)

The consequence of this is the need for explicitly creating *interior faces* with the appropriate attributes for both sides of each face; and this we do, although only for rendering environments that can support transparency. While the creation of interior faces would seem to be unnecessary whenever both sides of a boundary are fully opaque impling the boundary is never visible, any subsequent reduction in the opacity will reveal the interior boundaries. Thus we do not make this optimization.

Another geometric operation of considerable utility that we have incorporated is *collision detection*. While the temptation is to think of collision detection as merely intersection, we take the view that unlike intersection the returned value is essentially symbolic rather than geometric. Consider two "macro objects", molecules for instance, that are created from the union of many individual components, such as atoms and bonds. When we answer the collision question, we generate a list of all pairwise collisions between the components from the two macro objects. That is, if the component names (implemented, for example, as pointers to their representation) are considered to serve as the labels for nodes of a *collision graph*, then what we produce are the arcs of this graph. (The arcs are directed, going from a component in the first operand to a component in the second; thus the name space of the two operands need not be disjoint.)

The advantage of this symbolic approach is that it separates the mechanism of collision detection from the policy of what action to take as a consequence of any collisions. Below in the section Interactive Techniques, we describe two different policies for two different circumstances, neither of which is concerned with the geometry of the intersection. Such a separation is made affordable because we perform collision detection as a small additional overhead to any set operation, and as we discuss below in the section Modeling, we use union to create the model each time the tool-object moves. The cost for collision detection is small because the algorithm for any set operation recurses until at least one operand is reduced to a cell. If this is an in-cell, then the additional work is no more than searching the tree of the other operand for other in-cells (containing identifiers) and the appropriate collision arcs are generated. We gain some efficiency by maintaining an identifier at any internal node whose region contains only one identifier (a list of identifiers at internal nodes is another possibility). The arcs are maintained as a set; that is, there are no duplicates.

Affine Transformations

Affine transformations are an elementary geometric operation whose implementation has customarily been treated as synonymous with 4x4 matrices. However, notable benefits can be gleaned if they are treated in a more sophisticated way. An important motivation for this is the need to provide an interactive user with intuitive ways to modify an object using affine transformations. Traditionally, we think of transformations being applied in the order in which they are specified; this is typically achieved by maintaining a composite matrix so that a new transform modifies the composite. While this policy is indeed the right one in many situations, it is not ideal for interactive object modification.

Instead, we have found from subjective experience that the primitive transformations should be applied in the following order: scaling, shearing, rotation and translation. Note that the transforms which modify distance, i.e. scaling and shearing, appear before the rigid body transforms. Consequently, we maintain each of the four primitive transformations separately and multiply them together each time their composite value is demanded, replacing the previous composite matrix. So for instance, a rotation operation modifies only the rotation matrix rather than the composite matrix. An additional consideration is the fact that all linear transforms have fixed points: the origin plus possibly some subset of coordinate axes. Changing these fixed points is achieved by employing an additional change-of-basis matrix. Thus the composite matrix resulting from the product of the primitive transforms is pre-multiplied by the change-of-basis matrix and then post-multiplied by its inverse. One then has the freedom to choose any local coordinate system for an object.

A second advantage of implementing an affine transformation class as something different from a matrix is the ease with which optimizations can be performed transparently. The greatest disparity among the primitive transforms is the much greater speed at which translation can be effected. Since translation appears in an interactive environment with considerably greater frequency (moving objects about), detecting when an affine transformation is no more than a translation is an effective and simple optimization. Similar considerations also lead to determining when transformed normals need re-normalization. Rigid body transforms do not require this and symmetric scaling necessitates only divisions, not the calculation of the length of the new normal which entails computing a square root (profiling confirmed that indiscriminate re-normalization resulted in significant overhead). A final optimization is the generation of composite and inverse matrices only on demand rather than after every change to the affine transformation. The routines that modify affine transformations mark them as not up-to-date; and routines that use them for transforming points, normals and hyperplanes first requests an up-to-date composite and inverse matrices, which are generated only the first time they are demanded (if the affine transformation is not utilizing the pre-defined ordering of primitive transforms, the composite will already be up-to-date, so only the inverse needs to be computed).

Picking

A classic operation dating to the days of random-scan vector-refresh displays is that of picking an on-screen object using the current cursor position (originally a light pen). We achieve this by ray-casting. In screen-space the cursor position provides the initial x and y values of the ray origin, and its z-value is the screen-space position of the near clipping-plane (0 in our system). The ray direction is that of a screen-space projector whose length is the distance between the near and far clipping-planes in screen-space ([0 0 1] in our system). The ray, with $t_{min} = 0$ and $t_{max} = 1$, is then mapped to model-space by the inverse of the model-to-screen space transformation. We can now use the ray-tracing algorithm for partitioning trees [Naylor and Thibault 86] that finds the first intersection point within t_{min} and t_{max}. This returns the model-space intersection point, the surface normal, the near and far attributes, the classification of the ray segment from the origin to the intersection point (in or out), and most importantly the identifier. Generating an equivalent "pick report" using standard methods usually entails clipping the entire model to a tiny window. Our method reflects directly the nature of the query and is more efficient due to the partitioning tree's search structure.

Modeling

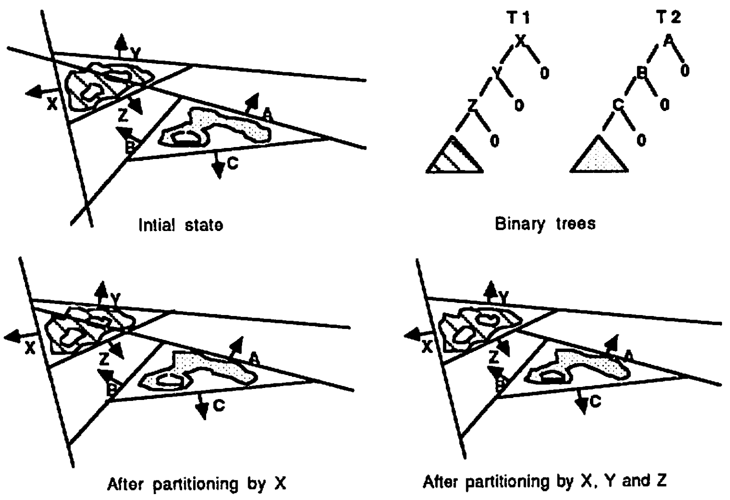
The representation of each user-level object entails three geometric sets, each represented by a partitioning tree. In particular, user-level objects are represented by an object-instance class containing the object definition, its value prior to the last set operation, an instance of the definition, and an affine transformation which determines the instance in terms of the definition (along with some minor state information).

As we mentioned in the user-view section, the model is a set of user-level objects one of which is distinguished as being the tool-object; the remaining objects we refer to as the *static-objects*. Any time a new tool-object is selected, a union of the static-objects is created as a distinct geometric set (represented by a single partitioning tree). A copy of each object's instance is used for this. The model is then the union of two sets: a copy of the static-objects and a copy of the tool-object. Any time the tool-object is modified, say by moving it, the previous model is discarded and a new one constructed; but note that the static-objects representation remains the same and so is reused as long as this set does not change (by set operations or picking a new tool-object).

Since we are using this in an interactive environment, these operations must be relatively quick. Two techniques are used to accelerate their execution. The first is the construction of good partitioning trees [Naylor 92]. This subject is somewhat complex, and so we will not address it here. But the essence is that we use expected case models to build partitioning trees that represent an object something like a sequence of approximations. An artifact of this process is the creation of a first level approximation that corresponds to the classic technique of bounding volumes. Figure 2 illustrates the effect of this when forming the union of two disjoint sets; in this case the computation is equivalent to forming the union of the bounding volumes, and so is very fast.

The second technique is a classic one: reference counts. Copying a tree has the effect initially of only incrementing the reference count to the root of the tree. Actual duplication of a tree node occurs only when it or its subtrees are modified. During set operations, any subtree of one tree lying within a cell of the other tree does not need to be duplicated in order for it to be included in the result.

Returning to figure 2, if the two trees were intended to be copies, only the first three nodes of each tree will be duplicated, while the subtrees indicated by filled triangles would remain untouched. Thus copying and discarding can require only sub-linear time. Reference counts are used as well for copying of polygons and attributes. One consequence of this for attributes is that the same allocated attributes will be shared by the object definition, its instance, and the copy of the instance used to form the model. Thus modifying the attributes of the object definition also achieves this modification for its instance and the copies of the instance present in the static-objects tree and in the model tree without the need for any additional work (other than rendering the new image).



Union of disjoint objects
Figure 2

Rendering

Given a partitioning tree representation of a collection of sets, for any viewing position a total visibility priority ordering of the sets can be generated by a single view-dependent traversal of the tree [Fuchs, Kedem and Naylor 81]. Since we have constructed a single partitioning tree representation of the entire model, we can solve the visible surface problem by performing the ordering operation on this tree.

There are several advantages to using the partitioning tree for visibility instead of the commonly used depth-buffer algorithm. First, the ordering is generated in model-space as opposed to the discrete, post-perspective screen-space. Thus, no depth-buffer is needed to represent this discrete space and surface continuity (coherence) is exploited to avoid performing ordering calculations for each pixel. This has contributed to the ability of our system to provide interactive solid geometry on workstations with no graphics hardware per se and requires only the presence of 2D convex polygon drawing routines. In addition, the information loss due to the perspective division using floating-point is avoided. Thus, for sets which intersect, or nearly intersect, their ordering will not vary incorrectly in the neighborhood of their intersection when small changes in the viewing position are made (this is especially a problem with polygons that are close and almost parallel). One consequence of this is that the edges of a polygon can be drawn "on top of" the polygon without any subsets of their discrete representation being occluded.

A second class of advantages of a visibility ordering arises from the desire to render non-refractive transparent objects and to perform anti-aliasing. Both of these operations can be implemented at the pixel level by blending the colors of two pixels. Given only two polygons (or pixels) p_1 and p_2, in which p_1 has color c_1 and opacity α_1 and occludes p_2 which has color c_2 and opacity α_2, then the resulting color is $c_{1,2} = (c_1 * \alpha_1) + (c_2 * \alpha_2) * (1 - \alpha_1)$ and opacity is $\alpha_{1,2} = \alpha_1 + (1-\alpha_1) * \alpha_2$. Anti-aliasing using a box filter can be treated analogously if the polygon is modeled as covering the entire pixel but with opacity equal to the area of the pixel covered by that polygon (this is in lieu of computing the visible surface at the sub-pixel level using masks). The correct use of this technique requires that blending occur only between pixels that are consecutive in the priority order, which cannot be accomplished with the standard depth-buffer algorithm. However, a multi-pass two-buffer algorithm is known [Mammen 89] where the number of passes equals 1 + the maximum number of overlapping transparent polygons. While transparency can be performed with either a far-to-near or near-to-far priority ordering of the partitioning tree, anti-aliasing requires the near-to-far ordering in order to avoid blending in occluded surfaces. Such an ordering is also needed for maintaining sub-pixels masks, if available, that are required for higher fidelity anti-aliasing, and for avoiding the calculations at fully occluded pixels (when $\alpha_1 = 1$) required by texture mapping or Phong shading.

Another major component of the rendering operation that can be achieved simply and efficiently using partitioning trees is clipping. View-volume clipping is nothing more than forming the intersection of the view-volume with the model. Thus, we generate a partitioning tree representation of the view volume and execute the standard partitioning tree intersection algorithm, but with the following optimization. Since the results from clipping will not, in an interactive environment, be used in any subsequent set operations, there is no need to produce the 3D geometry of the intersection explicitly. Only the intersection of the model's faces with the view volume is required. Consequently, the operation does not modify the tree representing the model and so avoids all copying accept for those faces that intersect the boundary of the view volume.

There are three advantages to partitioning tree clipping. First, the efficiency of set operations is enhanced by the expected case performance of the partitioning tree when thought of as a search structure. Roughly speaking, one might expect for **n** faces **O(log n)** operations instead of the **O(n)** of the standard implementation of clipping. For example, any object that is entirely inside or entirely outside the view-volume will be clipped by "visiting" only the top level nodes forming its bounding volume. Thus, culling is achieved without any explicit notion of culling just as there is no explicit notion of bounding volumes (and so no code to implement them); it arises from building good trees for efficient set operations. This contributes to the effectiveness of the system when used on standard workstations. Secondly, near-plane clipping does not reveal a polyhedron as an empty shell but rather maintains the semantics of solids. Not only are the faces of the view-volume lying within objects displayed, but they have the attributes of the region which they bound. (Note that b-reps do not typically have cells with attributes required by this.) This permits near-plane clipping to be used to provide cut-aways as a visualization aid. Thirdly, the partitioning tree clipping algorithm supports arbitrary polyhedral view volumes just as easily as the traditional truncated pyramid. This can be put to good use in an environment with multiple overlapping viewports with simultaneous renderings in each. Portions of a model occluded by another viewport can be clipped away by using the appropriate view-volume.

Interactive Techniques

Given the above capabilities, we have chosen several interactive techniques that enhance the usability of the system. One is the use of collision detection to provide highlighting via transparency. Forming the union of the tool-object with the static-objects, like any set operation, can generate a collision report. We can exploit this so that whenever the tool-object makes contact with any object in the scene, the contacted objects have their opacity reduced, and then subsequently restored when contact ceases. For rendering systems supporting transparency, this has the effect of not only indicating contact, but also allowing the user to see the tool-object as it moves through the interior of the contacted objects. For rendering systems without transparency, the reduction in opacity appears instead as a reduction in luminance, since the percentage of reflected light decreases with a decrease in opacity. In such environments, using the near-plane clipping as a cut-away can reveal the position of the tool-object as it moves through the interior. (Note that this visualization technique requires that of the tool-object's attributes take precedence over the static-objects when forming the model.) Cutaways also provide a means of inspecting the otherwise occluded interior parts of objects.

A second use of collision detection is for operand selection in the execution of a user specified set operation. The set operation is performed between the tool-object and only those objects with which it is in contact. This method is, from the user's perspective, simple and intuitive. Not having some method of selection is unacceptable since, for example, intersection between the tool-object and objects with which it is *not* in contact would annihilate those objects.

Those objects which are modified by a set operation will also have their center of mass and principal axes recomputed, after which their object definition is translated and rotated so that the center of mass and the principle axes form the object's local coordinate system while the instance reamains stationary. Thus affine transformations are always with respect to an objects "natural" coordinate system. This is very important if one wishes to maintain an intuitive sense of the effects of affine transformations when applied to objects whose geometry has been significantly modified by a set operation.

Picking provides a direct geometric method of tool-object selection, and implementing this with ray-casting makes the selection precise. Dragging of the tool-object is achieved by mapping the translation vector defined in screen-space into model-space. This requires that the screen-space vector have a depth, which the mouse does not provide. For this, we use the screen-space depth value of the picked surface point. As a consequence the picked position always tracks the cursor exactly (as long as the view is not changed). A similar operation is placing, in which the user picks a source point on the tool-object and then picks a target point on any other object, while during the interim changing the view if necessary to make the target point visible. The source point is then translated to the target point.

Finally, we use a variety of other minor techniques to improve usability. Tool-object translation and view rotation via mouse translation is always with respect to screen-space; so, for example, left-right mouse motion maps to horizontal motion in a plane parallel to the screen. We also adjust the view rotation rate as a function of window size so that the subjective sense of the rate remains constant. Symmetric scaling of the tool-object increases as a constant percentage of the current scale factor, while differential scaling changes by a constant step size independent of the current scaling (empirically ascertained preference). Delta changes generated by holding down buttons have a slight acceleration and are adjusted to compensate for varying frame update rates; otherwise they would be too slow for long update times and too fast for short ones. And shearing preserves basis vector length rather than volume; otherwise surface area increases unboundedly even though the volume remains the same.

Comparison to Buffer Algorithms

The ability to transform and merge partitioning trees quickly has obviated the original restriction in which trees were only created during a pre-processing phase. Since affine transformations and perspective projections (or more generally d+1 dimensional linear transformations) do not change a tree's structure, off-line generation of good trees leads to efficient merging of instances of these trees, resulting in reasonably good new trees. Thus, we have only relaxed, not abandoned, the original pre-processing context, and this is a crucial point. For the original idea was to exploit time-invariant properties of the geometry to reduce computation. Thus complex but affinely invariant objects can still have their trees created once as a pre-processing step. Interactively created objects can also have their trees improved by off-line reconstruction. However, this investment can now be reaped in a much less restrictive environment, since these trees can be transformed and merged.

As the system described in this paper illustrates, the pre-computation yields efficient set operations, collision detection, clipping, visible surface determination, transparency, anti-aliasing and picking. Let us compare these capabilities to the screen-space approaches for polyhedral objects which are extensions in the spirit of the z-buffer algorithm (as opposed to scan-line algorithms). In these methods, typically no pre-computation is utilized and each object is sampled independently. Screen-space evaluation of set operations requires a representation of the geometric model in every 1D sub-domain defined by the projector for each pixel (or sub-pixel); this, of course, is the same methodology as ray-cast evaluation of CSG models. Assuming retention of the scan-conversion modus operandi, as opposed to ray-tracing, this requires decomposing an object into a set of segments with attributes and identifier (for collision detection) instead of simply pixels. For convex primitives, this is not difficult, but for arbitrary polyhedra, no efficient scan-conversion type algorithm is known. The resulting segments must then be merged and evaluated as defined by the CSG tree (note that set operations require knowing the value of an operand in *every* projector, not just those intersecting the interior as would be discovered by scan-conversion). This will yield an ordered list of segments that can, for the purposes of transparency, be composited. The accuracy of this computation is affected not only by the sampling, but also by the rather considerable non-linear compression of the depth. This is in addition to the difference in accuracy of discrete screen space vs. continuous model space computations; a point of some importance in non-interactive geometric computation, such as interference detection and mass property calculations. Anti-aliasing requires all of this work to be performed at the sub-pixel level if a correct image is to be generated, adding a factor of 4 or 16 increase in the per pixel computation. Clipping is O(n), although this can be reduced by an initial culling of objects by comparing their bounding box to the view-volume. However, clipping unculled objects is still O(n) and the view volume is restricted to being convex, so that in a window system, subsets of the model occluded by another window would not be eliminated by clipping (however, solid clipping, i.e. cutaways, could be achieved with this proposed system). Picking has often been implement as an additional O(n) clipping step as opposed to our ~O(log n) ray-cast, although with this proposed schema, the necessary information would already be present in the buffer.

Having done all this, the result is an image identical to the one generated using partitioning trees. However, evaluation of set operations would be repeated for every frame at every sub-pixel, even though an object created using set operations may be affinely invariant for an arbitrarily large number of frames (not just seconds, but possibly days or years). So let us assume that screen-space algorithms are not used for set operations. Then for collision detection and transparency, we still must have the ordered set of segments for each sub-pixel, since we are effectively computing the union of the objects. The comparative cost of this operation suffers both from the absence of pre-processing acquired temporal-correlation and from the non-exploitation of spatial-coherence: instead of determining spatial relations between volumes, the same relation will be repeatedly computed for every ray that intersects those volumes (scan-line algorithms, of course, can exploit coherence). Parallelization of the buffer-based approaches is not necessarily a panacea, since partitioning tree algorithms can be parallelized as well. Only in circumstances where the presupposition of affine invariance over a sufficient period of time is violated, which is commonly the case for cartoon like animations, would the buffer-based approaches be arguably superior for polyhedral models.

References

[Chin and Feiner 89]
Norman Chin and Steve Feiner, "Near Real-Time Shadow Generation Using BSP Trees", **Computer Graphics** Vol. 23(3), pp. 99-106, (June 1980).

[Fuchs, Kedem, and Naylor 80]
Henry Fuchs, Zvi Kedem and Bruce Naylor, "On Visible Surface Generation by a Priori Tree Structures," **Computer Graphics** Vol. 14(3), pp. 124-133, (July 1980).

[Fuchs, Abrams, and Grant 83]
Henry Fuchs, Gregory Abrams and Eric Grant, "Near Real-Time Shaded Display of Rigid Objects", **Computer Graphics** Vol. 17(3), pp. 65-72, (July 1983).

[Mammen 89]
A. Mammen, "Transparency and Antialiasing Algorithms implemented with the Virtual Pixel Maps Technique," **IEEE CG&A** Vol. 9(4), pp. 43-55, (July 1989).

[Naylor and Thibault 86]
Bruce F. Naylor and William C. Thibault, "Application of BSP Trees to Ray-Tracing and CSG Evaluation," GIT-ICS 86/03, School of Information and Computer Science, Georgia Institute of Technology, (February 1986).

[Naylor 90]
Bruce F. Naylor, "SCULPT: an Interactive Solid Modeling Tool," Proceeding of *Graphics Interface* , pp. 138-148 (May 1990).

[Naylor, Amanatides and Thibault 90]
Bruce F. Naylor, John Amanatides and William C. Thibault, "Merging BSP Trees Yields Polyhedral Set Operations," **Computer Graphics** Vol. 24(4), pp. 115-124, (Aug. 1990).

[Naylor 92]
Bruce F. Naylor, "Constructing Good Partitioning Trees," manuscript in preparation.

[Schumacker et al 69]
R. A. Schumacker, R. Brand, M. Gilliland, and W. Sharp, "Study for Applying Computer-Generated Images to Visual Simulation," AFHRL-TR-69-14, U.S. Air Force Human Resources Laboratory (1969).

[Thibault and Naylor 87]
W. Thibault and B. Naylor, "Set Operations On Polyhedra Using Binary Space Partitioning Trees," **Computer Graphics** Vol. 21(4), pp. 153-162 (July 1987).

Fast Algorithms For Rendering Cubic Curves

Benjamin Watson
Larry F. Hodges
Graphics, Visualization, and Usability Center
College Of Computing
Georgia Institute Of Technology
Atlanta, GA 30332

Abstract

We present two integer-only algorithms to be used in tandem for rendering cubic functions and parametric cubic curves with rational coefficients. Analysis of execution speed of existing algorithms shows that our algorithms will match or outperform other current algorithms. Furthermore, while other existing algorithms can only handle curves shaped by rational coefficients by introducing some approximation error, our algorithms always choose the best approximation. Curves may have to be split before rendering, because each algorithm only handles curves with slope in a certain range. When plotting parametric curves, our algorithms may require more bits of representation for some integer variables than other existing algorithms.

Keywords: display algorithms, curve representations, parametric curves, raster graphics

1. Introduction

Computer scientists have been developing line and curve-rendering algorithms for over 25 years. Only recently, however, have efficient algorithms for the plotting of cubic curves begun to appear. This paper will propose and develop two fast, integer-only algorithms that can be used in tandem to render cubic curves with rational coefficients defined by the function

$$y = (A_n/A_d)x^3 + (B_n/B_d)x^2 + (C_n/C_d)x + (D_n/D_d). \qquad (1)$$

The algorithms are based on the midpoint method, described by Van Aken and Novak in [17] and below.

2. History And Existing Algorithms

J.E. Bresenham was the first to present a fast, integer-only line rendering algorithm in 1965 [1]. Research in line rendering since has seized on the periodic patterns shown by Bresenham's algorithm when viewed on a raster display as a means of improving algorithm speed [4,14].

Algorithms for rendering circles began to appear in the 1970s. Bresenham [2,3], Horn [7], and McIlroy [12] have all presented algorithms. Later, algorithms for rendering ellipses were published [9,15,16], and more recently, algorithms for the plotting of parabolas and hyperbolas were presented [13,15,18].

Algorithms for the rendering of cubic curves have only begun to appear in the last few years. In [11], Klassen presented two algorithms for rendering parametric cubic curves. First he identified the family of Bezier curves that are "worst-case", meaning that they are most likely to cause overflow during calculation. If $2h$ is screen length or width, Klassen asserted that "worst case" curves would have the four Bezier control points $[-h,5h,-5h,h]$ in at least one dimension, which would describe the one-dimensional parametric Bezier cubic $32ht^3 - 48ht^2 + 18ht - h$. Klassen called such curves S curves. Klassen then presented his two algorithms and outlined their relative speed and overflow restrictions for worst-case curves. Algorithm A uses a fixed-point representation of curve coordinates, and thus incorporates an inherent level of error. However, it is fast and has a liberal overflow restriction. Algorithm B divides forward differences into integer and fractional parts, providing perfect accuracy. But it is slower than algorithm A, and can only take 1024 parametric steps if overflow is to be avoided with 32-bit words. Both algorithms allow arbitrary step size and do not restrict curve segments to certain slope octants. Both can be used with non-integer coefficients. However, use of such coefficients with algorithm B would eliminate its perfect accuracy.

In [10], Klassen studied the use of these two algorithms with cubic spline curves. He envisioned the use of the algorithms with adaptive forward differencing [6,8], which dynamically adjusts step size as a curve is plotted.

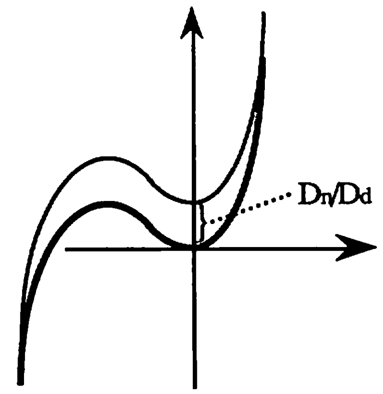


Figure 1: Elimination of the constant term can be compensated for by a translation.

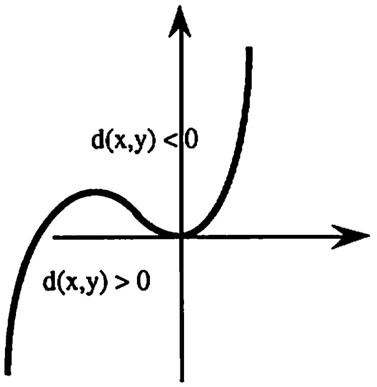


Figure 2: If the decison function is evaluated at a point above the curve, it is negative. Otherwise it is positive.

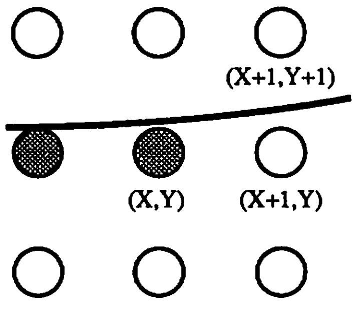


Figure 3: If |slope| < 0 and X and Y plotting directions are positive the candidate points are (X+1,Y) and (X+1, Y+1).

Table 1. The candidate and midpoints used depend on curve slope and X and Y plot direction. Points are listed in clockwise order.

Y Plot Dir	X Plot Dir	|Slope|	Candidate Points	Midpoints
+	+	< 1	(X+1,Y); (X+1,Y+1)	(X+1,Y+1/2)
+	+	>= 1	(X+1,Y+1); (X,Y+1)	(X+1/2,Y+1)
+	-	>= 1	(X,Y+1); (X-1,Y+1)	(X-1/2,Y+1)
+	-	< 1	(X-1,Y+1); (X-1,Y)	(X-1,Y+1/2)
-	+	< 1	(X-1,Y); (X-1,Y-1)	(X-1,Y-1/2)
-	+	>= 1	(X-1,Y-1); (X,Y-1)	(X-1/2,Y-1)
-	-	>= 1	(X,Y-1); (X+1,Y-1)	(X+1/2,Y-1)
-	-	< 1	(X+1,Y-1); (X+1,Y)	(X+1,Y-1/2)

Simultaneous to Klassen, Chang et al. [5] developed an algorithm similar to Klassen's algorithm B that also could be used with adaptive forward differencing. Differences between the two algorithms are minor.

3. Preliminaries

3.1. Elimination Of The Constant Term D_n/D_d

Since the last term (D_n/D_d) in (1) does not change the shape of the curve, we can render the curve described by instead rendering the curve

$$y = (A_n/A_d)x^3 + (B_n/B_d)x^2 + (C_n/C_d)x \qquad (2)$$

with a compensating translation (see figure 1). Note that if the (D_n/D_d) rational coefficient is not an integer, translation of the Y coordinate at plotting by round(D_n/D_d) alone will not necessarily produce the best approximation of the curve. In section 4 we will discuss a method of compensating for this inaccuracy.

3.2. The Midpoint Method

The midpoint method, described by Van Aken and Novak in [17], requires the incremental evaluation of a decision function that indicates which of two candidate pixels should be chosen for rendering. If the equation for a curve is $y = f(x)$, then the decision function has the form $d(x,y) = f(x) - y$. Notice that this function will have a different sign on each side of the curve f(x) (see figure 2).

Where do we evaluate this function? This depends on the slope of the curve. If $-1 < f'(x) < 1$ and we are plotting in positive X and Y directions, then if we have just plotted the point (X,Y), the two candidate points for plotting are (X+1,Y) and (X+1,Y+1). (See figure 3. Table 1 shows a complete list of candidate points.) The midpoint method evaluates the decision function at the midpoint between the candidate pixels. In our example, this midpoint is (X+1,Y+1/2), and thus we evaluate d(X+1,Y+1/2). (See figure 4. Table 1 shows a complete list of midpoints.) We

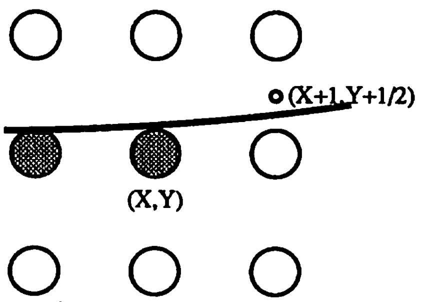


Figure 4: If |slope| < 0 and the X and Y plotting directions are positive, the midpoint is (X+1,Y+1/2).

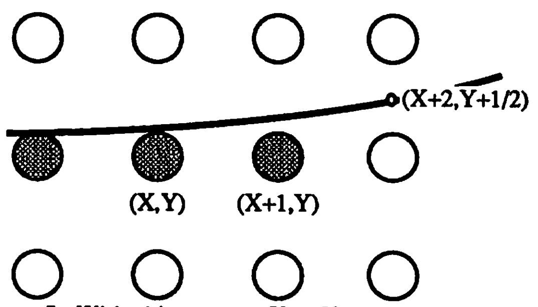


Figure 5: With this curve, (X+1,Y) would be plotted, and (X+2,Y+1/2) used as the next midpoint.

will call the decision function the "decision variable" when it is evaluated at a midpoint.

Since the sign of the decision function d(x,y) corresponds to a specific side of f(x), the sign of decision variable d(X+1,Y+1/2) indicates the side of f(x) on which the midpoint lies, and also which of the candidate pixels lies closer to the curve being plotted, f(x). In figure 4, the sign of the decision variable is negative, so the lower candidate pixel is chosen for plotting.

Once the next pixel is chosen and plotted, the decision function must be evaluated at the next midpoint to allow the plotting of the next pixel. In our example (figure 5), the appropriate decision variable would be d(X+2,Y+1/2).

3.3 Forward Differencing

Simple evaluation of the decision function at each successive midpoint would be computationally expensive. Fortunately, there is a method of incremental function evaluation, called forward differencing, which is uniquely suited to our needs. This method, which was known to Newton, involves the initialization of several difference values that may be added together to produce the value of a function at a certain point. These difference values are then themselves incrementally evaluated, to prepare for the next evaluation of the original function. Note that mulitiplication is only required for function and difference initialization. Furthermore, if all function coefficients are integers, no floating point addition is required.

As an example, consider the simple function f(x) = 2x + 1. f(x+1) differs from f(x) only by the constant difference value 2. By successively adding 2 to an initial value for f(x), we could incrementally calculate the value of f(x) at integer intervals on X. For the higher-order function $g(x) = x^2$, the binomial expansion $g(x+1) = (x+1)^2 = x^2 + 2x + 1$ gives us the first-order difference value 2x + 1 for an integer interval. Since this difference value is also dependent on X, it must also be subjected to forward differencing, as already discussed. Thus the incremental calculation of $g(x) = x^2$ would require two additions per integer interval.

4. The RunRise Algorithm

Let us first find the decision function d(x,y) for equation (2) when -1 < f'(x) < 1. In this case, the X component of f'(x) is larger than the Y component: the curve "runs" faster than it "rises." We will label segments of f(x) where this condition holds true "RunRise." Since we will only make use of the sign of our decision function, we multiply our cubic function (2) by $2A_dB_dC_d$ to increase efficiency by eliminating the floating point division calculations. To conserve space, we use the shorthand $A_i = A_nB_dC_d$, $B_i = B_nA_dC_d$, $C_i = C_nA_dB_d$, and $D_i = A_dB_dC_d$, in the rest of this paper and the equation below. We assume without loss of generality that D_i (and thus the denominators A_d, B_d, and C_d) are positive:

$$2D_iy = 2A_ix^3 + 2B_ix^2 + 2C_ix. \qquad (3)$$

We will find it useful to plot in both positive and negative X and Y directions. Our direction-flexible decision variable d(x±1,y±1/2) is then

$$2A_i(x\pm1)^3 + 2B_i(x\pm1)^2 + 2C_i(x\pm1) - 2D_i(y\pm1/2) \qquad (4)$$

where ± is positive if we are plotting in a positive direction, negative otherwise.

We must evaluate (4) incrementally as we plot the RunRise portion of f(x). To avoid computationally complex multiplications, we will use forward differencing. The difference constant d_{0y} is the difference between d(x,y) evaluated at the "current" Y coordinate, and d(x,y) evaluated at the "next" Y coordinate:

$$\begin{aligned} d_{0y} &= d(x\pm1,y\pm3/2) - d(x\pm1,y\pm1/2) \\ &= 2D_i(y\pm3/2) - 2D_i(y\pm1/2) \\ &= \pm2D_i. \end{aligned}$$

This difference constant is used to update the decision variable when a "Y step" is made -- that is, when the last plotted pixel differs from the previously plotted one in its Y coordinate.

Because the decision variable (4) is a third-order function in X, updating it when an X step is made is more complex. The second order difference function $d_2(x)$ is the difference between the decision variable at the current X coordinate and the next X coordinate:

$$\begin{aligned} d_2(x) &= d(x\pm1,y\pm1/2) - d(x,y\pm1/2) \\ &= 2A_i(x\pm1)^3 + 2B_i(x\pm1)^2 + 2C_i(x\pm1) - 2A_ix^3 \\ &\quad - 2B_ix^2 - 2C_ix \\ &= 2A_i(x^3\pm3x^2+3x\pm1) + 2B_i(x^2\pm2x+1) + 2C_i(x\pm1) \\ &\quad - 2A_ix^3 - 2B_ix^2 - 2C_ix \\ &= 2A_i(\pm3x^2+3x\pm1) + 2B_i(\pm2x+1) \pm 2C_i \\ &= \pm6A_ix^2 + (6A_i\pm4B_i)x + (\pm2A_i+2B_i\pm2C_i). \end{aligned}$$

$d_2(x)$ must itself be subjected to forward differencing. The first order difference function $d_1(x)$ is the difference between the value of $d_2(x)$ at the current and next X coordinates:

$$\begin{aligned} d_1(x) &= d_2(x\pm1) - d_2(x) \\ &= \pm6A_i(x\pm1)^2 + (6A_i\pm4B_i)(x\pm1) - \pm6kA_ix^2 \\ &\quad - (6A_i\pm4B_i)x \\ &= \pm6A_i(x^2\pm2x+1) + (6A_i\pm4B_i)(x\pm1) - \pm6A_ix^2 \\ &\quad - (6A_i\pm4B_i)x \\ &= \pm6A_i(\pm2x+1) + (\pm6A_i+4B_i) \\ &= 12A_ix + (\pm12A_i+4B_i). \end{aligned}$$

Finally, $d_1(x)$ must be subjected to forward differencing. The difference between $d_1(x)$ evaluated at the current and next X coordinates gives us constant d_{0x}:

$$\begin{aligned} d_{0x} &= d_1(x\pm1) - d_1(x) \\ &= 12A_i(x\pm1) - 12A_ix \\ &= \pm12A_i. \end{aligned}$$

Stepping one pixel at a time, and using the global declarations below, we can construct a direction-flexible algorithm for plotting the RunRise segments of cubic curves. Figures 6 and 7 show the core of this algorithm in C. For brevity's sake, we have removed the variable and function declarations (all variables are long integers). The initialization of A_i, B_i, C_i and D_i is not shown -- it is common to all curve segments.

In its loop, the RunRise algorithm performs 4 additions for each step in X, 2 additions for each step in Y. This gives a cost of

$$4a|\mathtt{XEnd}-\mathtt{X}| + 2a|\mathtt{YEnd}-\mathtt{Y}|,$$

where (X,Y) and (XEnd,YEnd) are the endpoints of the plotted curve segment, and a the cost of one addition operation.

Earlier, we noted that compensating for the elimination of the non-integer constant with translation will not necessarily produce the best approximation of the curve. Adding an appropriately signed round(D_i * ((D_n/D_d) mod 1)) to the initial decision variable will simulate a fractional Y step and improve accuracy.

5. The RiseRun Algorithm

Now let us find the decision variable needed when $|f'(x)| > 1$. In this case, it is the Y component of f'(x) that is larger, so the curve will "rise" faster than it "runs." We will label segments of f(x) where this condition holds true "RiseRun."

As is clear from table 1, the direction-flexible midpoint decision variable is $d(x\pm1/2,y\pm1)$. Expanded, this is

$$8A_i(x\pm1/2)^3 + 8B_i(x\pm1/2)^2 + 8C_i(x\pm1/2) - 8D_i(y\pm1). \quad (5)$$

We have used the constant $8D_i$ rather than $2D_i$ in (5) to allow the integer performance of the half step $x\pm1/2$. (5) reduces as follows:

$$\begin{aligned} &= 8A_i(x^3\pm3/2x^2+3/4x\pm1/8) + 8B_i(x^2\pm x+1/4) \\ &\quad + 8C_i(x\pm1/2) - 8D_i(y\pm1) \\ &= 8A_ix^3 + (\pm12A_i+8B_i)x^2 + (6A_i\pm8B_i+8C_i)x \\ &\quad + (\pm A_i+2B_i\pm4C_i) - 8D_i(y\pm1) \end{aligned}$$

We use this all-integer equation to initialize our decision variable. The following forward stepping increments allow us to update that decision variable:

$$\begin{aligned} d_2(x) &= d(x\pm3/2,y\pm1) - d(x\pm1/2,y\pm1) \\ &= 8A_i(x\pm3/2)^3 + 8B_i(x\pm3/2)^2 + 8C_i(x\pm3/2) \\ &\quad - 8A_i(x\pm1/2)^3 - 8B_i(x\pm1/2)^2 - 8C_i(x\pm1/2) \\ &= 8A_i(x^3\pm9/2x^2+27/4x\pm27/8) + 8B_i(x^2\pm3x+9/4) \\ &\quad + 8C_i(x\pm3/2) \\ &= 8A_i(x^3\pm3/2x^2+3/4x\pm1/8) - 8B_i(x^2\pm x+1/4) \\ &\quad - 8C_i(x\pm1/2) \\ &= \pm24A_ix^2 + (48A_i\pm16B_i)x + (\pm26A_i+16B_i\pm8C_i) \end{aligned}$$

$$\begin{aligned} d_1(x) &= d_2(x\pm1) - d_2(x) \\ &= \pm24A_i(x\pm1)^2 + (48A_i\pm16B_i)(x\pm1) - \pm24A_ix^2 \\ &\quad - (48A_i\pm16B_i)x \\ &= \pm24A_i(x^2\pm2x+1) + (48A_i\pm16B_i)(x\pm1) - \pm24A_ix^2 \\ &\quad - (48A_i\pm16B_i)x \\ &= \pm24A_i(\pm2x+1) + (\pm48A_i+16B_i) \\ &= 48A_ix + (\pm72A_i+16B_i) \end{aligned}$$

```
XDec2Const1 = Ai<<1;                               /* Used to init 2nd order diffc function
*/
XDec0Const = (XDec2Const1<<2) + (XDec2Const1<<1); /* Used to init diffc constant */
XDec2Const2 = Bi<<1;                               /* Used to init 2nd order diffc function
*/
XDec1Const = XDec2Const2<<1;                       /* Used to init 1st order diffc function
*/
XDec2Const3 = Ci<<1;                               /* Used to init 2nd order diffc function
*/
DecConst = Di;                                     /* Used to init decision variable */
YDecConst = DecConst<<1;                           /* Difference constant for a Y step */

Templ = XDec2Const1*X;                             /* Used several times to save multiplies
*/
if (X < XEnd) {                                    /* If plotting in positive X direction */
   XStep = 1;                                      /* Set X increment */
   XDec0 = XDec0Const;                             /* Difference constant for an X step */
   XDec1 = (Templ<<2) + (Templ<<1)                 /* Init 1st order diffc function */
      + XDec0Const + XDec1Const;
   XDec2 = ((Templ<<1) + Templ                     /* Init 2nd order diffc function */
      + (XDec0Const>>1) + XDec1Const)*X
      + XDec2Const1 + XDec2Const2 + XDec2Const3;
} else {                                           /* If plotting in negative X direction */
   XStep = -1;                                     /* Set X increment */
   XDec0 = -XDec0Const;                            /* Diffc constant for an X step */
   XDec1 = (Templ<<2) + (Templ<<1)                 /* Init 1st order diffc function */
      - XDec0Const + XDec1Const;
   XDec2 = (-(Templ<<1) - Templ                    /* Init 2nd order diffc function */
      + (XDec0Const>>1) - XDec1Const)*X
      - XDec2Const1 + XDec2Const2 - XDec2Const3;
} /* end if */

Dec = ((Templ + XDec2Const2)*X + XDec2Const3)*X;  /* Init decision variable */
if (Y < YEnd) {                                    /* If plotting in positive Y direction */
   YStep = 1;                                      /* Set Y increment */
   Dec = Dec + XDec2 - YDecConst*Y - DecConst;     /* Final decision variable init'zation */
} else {                                           /* If plotting in negative Y direction */
   YStep = -1;                                     /* Set Y increment */
   XDec0 = -XDec0;                                 /* Negate X differences */
   XDec1 = -XDec1;
   XDec2 = -XDec2;
   Dec = -Dec + XDec2 + YDecConst*Y - DecConst;    /* Final decision variable init'zation */
} /* end if */
```

Figure 6: RunRise algorithm initialization. (X,Y) and (XEnd,YEnd) are segment endpoints.

```
if (X == XEnd)                                     /* If degenerate curve, */
   {LinPlot(X,Y,XEnd,YEnd);                        /* Plot a line */
else                                               /* Otherwise, */
   for (X=X; X<=XEnd; X=X+XStep) {                 /* For each X in the curve */
      Plot(X,Y);                                   /* Plot a point */
      XDec2 = XDec2 + XDec1;                       /* Update the 2nd order diffc */
      XDec1 = XDec1 + XDec0;                       /* Update the 2st order diffc */
      if (Dec > 0) { /* perform Y step */          /* If must perform Y step */
         Y = Y + YStep;                            /* Adjust Y accordingly */
         Dec = Dec + XDec2 - YDecConst;            /* Update the decision var accordingly */
      } else Dec = Dec + XDec2;                    /* If no Y step, update dec var accdgly */
   } /* end for */
```

Figure 7: RunRise algorithm loop. (X,Y) and (XEnd,YEnd) are curve segment endpoints.

$$\begin{aligned} d_{0x} &= d_1(x\pm1) - d_1(x) \\ &= 48A_i(x\pm1) - 48A_ix \\ &= \pm48A_i \end{aligned}$$

$$\begin{aligned} d_{0y} &= 8D_if(x\pm1/2,y\pm2) - 8D_if(x\pm1/2,y\pm1) \\ &= 8D_i(y\pm2) - 8D_i(y\pm1) \\ &= \pm8D_i. \end{aligned}$$

Figures 8 and 9 show the RiseRun algorithm. Again, variable declarations and global initializations are not shown.

The RiseRun algorithm, like the RunRise algorithm, performs 4 additions for each step in X, 2 additions for each step in Y. Thus algorithm cost is

$$4a|\texttt{XEnd-X}| + 2a|\texttt{YEnd-Y}|.$$

Note, however, that |YEnd-Y| will in this case always be greater than |XEnd-X|.

6. Use Of The Algorithms In Tandem

Because each of the RunRise and RiseRun algorithms only function for curve segments that have slope within a certain range, plotting a cubic curve with these algorithms will typically require that the curve be split into segments which satisify the algorithms' slope range requirements. Algorithm initialization must then be performed for each curve segment, increasing overhead.

The actual curve splitting itself will also increase overhead. Since each algorithm takes as input only function constants and segment endpoints, splitting essentially involves the location of appropriate segment endpoints. For the unoptimized algorithms, these endpoints will be the points on the curve at which the conditions $|f'(x)| = 1$ and $f'(x) = 0$ hold true. Locating these points will involve floating point arithmetic. The appropriate algorithm must then be called for each segment. In the worst case, a curve will have to be split into seven such segments. However, use of the algorithms with spline and Bezier curves would typically require the splitting of curves into only two or three segments.

7. Overflow

Care must be taken to avoid overflow when using these algorithms. Below, we provide overflow analysis for the initialization and looping portions of each of the algorithms.

We begin with the RunRise algorithm. During initialization, the largest intermediate value that must be handled is the first initialization of the decision variable, `((Templ + XDec2Const2)*X + XDec2Const3)*X`. This represents $2A_ix^3 + 2B_ix^2 + 2C_ix$. For ease of algorithm use and analysis, we define i such that each of the coefficients $|A_i|$, $|B_i|$, $|C_i|$ and $|D_i|$ are less than i. The screen space variable $|x|$ has the maximum value $w/2$, where w is screen width. Thus in the worst case, the decision variable will take on the intermediate value $|iw(w^2/4 + w/2+ 1)|$. To represent this value without overflow, the condition

$$|iw(w^2/4 + w/2+ 1)| < 2^{bits-1},$$

where *bits* is the number of bits available for representation, must hold true. As an example, it is reasonable to expect screen width w to be less than 1280. We then have

$$\begin{aligned} 1280i\,(320^2 + 640+ 1) &< 2^{bits-1} \\ 1280i\,(103041) &< 2^{bits-1} \\ |i| &< 2^{bits-1}/131892480. \end{aligned}$$

If a 32-bit word is to be used for representation, $bits = 32$ and $|i|$ must be less than or equal to 16. Clearly, this is too restrictive. If instead 64 bits are used during initialization, we have $|i| < 3.496549627e+10$. Considering that $\lfloor\log(3.496549627e+10)\rfloor$ is 35, this is quite reasonable.

The algorithm changes the state of 5 variables while looping: `X`, `Y`, `Dec`, `XDec1`, and `XDec2`. `X` and `Y` are screen space variables, and thus will not overflow unless any screen coordinate is larger than 2^{bits-1} in magnitude -- an unlikely event, given the present state of raster technology. `XDec1` represents the difference function $d_1(x) = d_2(x\pm1) - d_2(x)$, and `XDec2` the difference function $d_2(x)$. By definition, then, `XDec2` will always be larger than `XDec1` in magnitude.

`Dec` represents the decision function $d(x,y)$ evaluated at the midpoint $(X\pm1, Y\pm1/2)$. When curve slope $|f'(x)| < 1$ (the RunRise case), the midpoint method guarantees that this point will be a distance d of at most 1/2 in Y from the point $(X\pm1,f(X\pm1))$ (see again figure 4 and table 1). Thus we have

$$d = |f(X\pm1) - (Y\pm1/2)| \le 1/2.$$

Scaling by $2D_i$ gives

$$2D_id = |2D_if(X\pm1) - 2D_i(Y\pm1/2)| \le D_i.$$

Note now that $2D_if(X\pm1) - 2D_i(Y\pm1/2)$ is equivalent to the decision variable (4). Thus we have $|d(x\pm1,y\pm1/2)| = |\texttt{Dec}| \le D_i$.

`XDec2` represents the difference function $d_2(x) = d(x\pm1,y\pm1/2) - d(x,y\pm1/2) = g(x\pm1) - g(x)$, where $g(x)$ is the function (3). We define the difference $d_2'(x) = d_2(x)/2D_i = f(x\pm1) - f(x)$, where $f(x)$ is the function (2). In the RunRise case, $|f'(x)| < 1$, which implies that $d_2'(x) = |f(x\pm1) - f(x)| \le 1$, and then it follows that $|\texttt{XDec2}| = |d_2(x)| \le 2D_i$. This allows us to conclude that representing the RunRise looping variables requires only the fulfillment of the almost trivial inequality

```
XDec0Const = (Ai<<5) + (Ai<<4);                  /* Used to init diffc constant */
XDec2Const = XDec0Const>>1;                      /* Used to init 2nd order diffc fction */
XDec1Const = XDec2Const + (Bi<<4);               /* Used to init 1st order diffc fction */
YDecConst = Di<<3;                               /* Difference constant for a Y step */
XDecConst = Ci<<3;                               /* Used to init 1st/2nd ord diffc fcts */

Templ = (Ai<<3)*X;                               /* Used to save multiplies */
if (X < XEnd) {                                  /* If plotting in positive X direction */
   XStep = 1;                                    /* Set X increment */
   XDec0 = XDec0Const;                           /* Difference constant for an X step */
   XDec1 = (Templ<<2) + (Templ<<1)               /* Init 1st order diffc function */
      + XDec0Const + XDec1Const;
   XDec2 = ((Templ<<1) + Templ                   /* Init 2nd order diffc function */
      + XDec1Const + XDec2Const)*X
      + XDec1Const + (Ai<<1) + XDecConst;
   Dec = ((Templ + (XDec1Const>>1))*X           /* Init decision variable */
      + (XDec1Const>>1) - (XDec2Const>>2)
      + XDecConst)*X + Ai + (Bi<<1)
      + (Ci<<2);
} else {                                         /* If plotting in negative X direction */
   XStep = -1;                                   /* Set X increment */
   XDec0 = -XDec0Const;                          /* Diffc constant for an X step */
   XDec1 = (Templ<<2) + (Templ<<1)               /* Init 1st order diffc function */
      - (XDec0Const<<1) + XDec1Const;
   XDec2 = (-(Templ<<1) - Templ                  /* Init 2nd order diffc function */
      + XDec0Const + XDec2Const
      - XDec1Const)*X + XDec1Const
      - XDec0Const - (Ai<<1) - XDecConst;
   Dec = ((Templ + (XDec1Const>>1)               /* Init decision variable */
      - XDec2Const)*X - (XDec1Const>>1)
      + XDec2Const - (XDec2Const>>2)
      + XDecConst)*X - (Ci<<2) + (Bi<<1)
      - XDecConst;
} /* end if */

if (Y < YEnd) {                                  /* If plotting in positive Y direction */
   YStep = 1;                                    /* Set Y increment */
   Dec = Dec - YDecConst*Y - YDecConst;          /* Final decision variable init'zation */
} else {                                         /* If plotting in negative Y direction */
   YStep = -1;                                   /* Set Y increment */
   XDec0 = -XDec0;                               /* Negate X differences */
   XDec1 = -XDec1;
   XDec2 = -XDec2;
   Dec = -Dec + YDecConst*Y - YDecConst;         /* Final decision varaible init'zation */
} /* end if */
```

Figure 8: RiseRun algorithm initialization. (X,Y) and (XEnd,YEnd) are segment endpoints.

```
if (Y == YEnd)                                   /* If degenerate curve, */
   {LinPlot(X,Y,XEnd,YEnd);                      /* Plot a line */
else                                             /* Otherwise, */
   for (Y=Y; Y<=YEnd; Y=Y+YStep) {               /* For each Y in the curve */
      Plot(X,Y);                                 /* Plot a point */
      if (Dec < 0) { /* perform X step */        /* If must perform X step */
         X = X + XStep;                          /* Adjust X accordingly */
         Dec = Dec + XDec2 - YDecConst;          /* Update the dec variable accordingly */
         XDec2 = XDec2 + XDec1;                  /* Update the 2nd order diffc */
         XDec1 = XDec1 + XDec0;                  /* Update the 1st order diffc */
      } else Dec = Dec - YDecConst;              /* If no X step, update dec var accdgly */
   } /* end for */
```

Figure 9: RiseRun algorithm loop. (X,Y) and (XEnd,YEnd) are curve segment endpoints.

$$2D_i < 2^{bits-1}.$$

Then if *bits* = 32, we must have $D_i < 2^{30}$ to loop in the RunRise algorithm without overflow.

The overflow analysis of initialization for the RiseRun algorithm is similar to the RunRise initialization analysis. The largest intermediate value calculated during initialization is the partial sum of the decision variable $8A_ix^3 + (\pm 12A_i+8B_i)x^2 + (6A_i\pm 8B_i+8C_i)x + (\pm A_i+2B_i\pm 4C_i)$. Here the worst case value is $|i\,(w^3 + 5w^2 + 12w + 7)|$, giving us the inequality

$$|i\,(w^3 + 5w^2 + 12w + 7)| < 2^{bits-1}$$

if overflow is to be avoided. Since this inequality is clearly more restrictive than the inequality required for RunRise initialization, the use of more than 32 bits is appropriate.

Because the condition $|f'(x)| < 1$ does not hold for RiseRun curves, overflow analysis of the RiseRun looping section will differ significantly from the similar RunRise analysis. The unoptimized RiseRun algorithm changes the same five variables as the unoptimized RunRise algorithm. We can again conclude that the overflow restrictions required by `Dec` and `XDec2` will most seriously affect program utility.

In the RiseRun algorithm, $d_2(x)/2D_i = f(x\pm 3/2) - f(x\pm 1/2)$. But since $|f'(x)| \geq 1$, we cannot conclude that $|f(x\pm 3/2) - f(x\pm 1/2)| \leq 1$ and $|\texttt{XDec2}| \leq 8D_i$. Theoretically, the difference $|f(x\pm 3/2) - f(x\pm 1/2)|$ could be infinite. Clearly, however, a curve that fulfilled such a condition would have infinite slope, and f(x) would then simply describe a vertical line. In fact, any curve segment that fulfills the condition `XEnd` = `X` would for our purposes be a vertical line, and would be most quickly rendered by a primitive line plotting routine. Since our RiseRun algorithm captures such cases, we can guarantee that `XEnd` ≠ `X` and thus that $|f(x\pm 3/2) - f(x\pm 1/2)| \leq h$, where *h* is screen height. This allows us to conclude that $|\texttt{XDec2}| = d_2(x) = |8D_if(x\pm 3/2) - 8D_if(x\pm 1/2)| \leq 8D_ih$, and gives us the restriction

$$8D_ih \leq 2^{bits-1}.$$

The RiseRun decision variable $d(x\pm 1/2, y\pm 1)$, like the RunRise decision variable, is proportional to a distance. We'll call this distance $d' = f(X\pm 1/2) - (Y\pm 1)$. However, because $|f'(x)| \geq 1$ for RiseRun curves, $|d'|$ has the wider range $[0,h]$, which implies that $|d(x\pm 1/2,y\pm 1)|$ has the proportional range $[0,8D_ih]$, and that $|\texttt{Dec}| \leq 8D_ih$. Thus the above overflow restriction for `XDec2` also applies to `Dec`.

If *bits* = 32 and *h* = 1024, we have

$$2^{13}\,D_i < 2^{31}$$
$$D_i < 2^{18}.$$

8. Algorithm Use With Parametric Curves

While the algorithms presented in this paper were designed for the direct plotting of cubic functions, they can be used, with slight modifications, to plot parametric curves.

Typically, parametric curves are plotted by making both the X and Y coordinates functions of a parameter *t*, which may take on values in the range [0,1]. Our algorithms could be used to plot such curves by running them twice: once to obtain X coordinates, and once to obtain Y coordinates. In both cases, the algorithm variable `X` would contain the current value of the parameter *t*. The variable `Y` would contain an X or Y coordinate. Below, we present overflow analysis only for the parametric equation x(*t*). For overflow restrictions for y(*t*), simply substitute *h* for *w*.

Since our algorithms can only use a step size of 1, `X` cannot, like t, take on values only in the range [0,1]. Instead, we could let it take on integer values in the range [0,*n*], where the even number *n* is the number of parametric steps desired. If we have the parametric equation

$$x(t) = At^3 + Bt^2 + Ct, \tag{6}$$

t in [0,1], the equation

$$x(t') = A(t'^3/n^3) + B(t'^2/n^2) + C(t'/n),$$

t' in [0,*n*], would describe the same coordinates. Thus the parametric version of the RunRise decision variable (4) is

$$A_i(t'\pm 1)^3 + B_in(t'\pm 1)^2 + C_in^2(t'\pm 1) - D_in^3(X\pm 1/2).$$

Note that we have not scaled the parametric decision variable by 2 because we know that *n* is an even number. The RunRise parametric differences are:

$$\begin{aligned}
d_2(t') &= A_i(t'\pm 1)^3 + B_in(t'\pm 1)^2 + C_in^2(t'\pm 1) - A_it'^3 - B_int'^2 \\
&\quad - C_in^2t' \\
&= A_i(t'^3\pm 3t'^2+3t'\pm 1) + B_in(t'^2\pm 2t'+1) + C_in^2(t'\pm 1) \\
&\quad - A_it'^3 - B_int'^2 - C_in^2t' \\
&= A_i(\pm 3t'^2+3t'\pm 1) + B_in(\pm 2t'+1) \pm C_in^2 \\
&= \pm 3A_it'^2 + (3A_i\pm 2B_in)t' + (\pm A_i+B_in\pm C_in^2)
\end{aligned}$$

$$\begin{aligned}
d_1(t') &= \pm 3A_i(t'\pm 1)^2 + (3A_i\pm 2B_in)(t'\pm 1) - \pm 3A_it'^2 \\
&\quad - (3A_i\pm 2B_in)t' \\
&= \pm 3A_i(\pm 2t'+1) \pm (3A_i\pm 2B_in) \\
&= 6A_it' + (\pm 6A_i+2B_in)
\end{aligned}$$

$$d_{0t'} = \pm 6A_i$$

For the parametric versions of our algorithms, we only present overflow restrictions for the looping sections. Analysis for the parametric RunRise variables `XDec2` and

Dec is quite similar to the analysis for the identically named non-parametric RunRise variables, and gives the overflow restriction

$$D_i n^3 < 2^{bits-1}.$$

If $bits = 32$ and $n = 256$, we have $D_i \le 128$.

Remember that t' refers to a step number rather than an absolute screen location. This enables us to use a low value of n, 128, and still know that the algorithm will function over the entire screen. If we were to make n smaller, the upper limit for $|D_i|$ would increase. If we had to plot a curve segment that required more than n steps, we could simply split the curve into several sub-segments of a more managable size.

Since many parametric curves interpolate or are controlled by points chosen on a computer screen, it is often the case that the coefficients A, B, C, and D in (6) are integers, not rational. In such cases, A_i, B_i and C_i are equal to A_n, B_n, and C_n. Most important, however, is the observation that $D_i = 1$. In such cases, our RunRise overflow restriction above becomes

$$n^3 < 2^{bits-1}.$$

If $bits = 32$, we have $n \le 1290$.

The parametric version of the RiseRun decision variable (5) is

$$8A_i(t'\pm 1/2)^3 + 8B_i n(t'\pm 1/2)^2 + 8C_i n^2(t'\pm 1/2) - 8D_i n^3(X\pm 1).$$

The RiseRun parametric differences are:

$$d_2(t') = \pm 24A_i t'^2 + (48A_i \pm 16B_i n)t' + (\pm 26A_i + 16B_i n \pm 8C_i n^2)$$

$$d_1(t') = 48A_i t' + (\pm 72A_i + 16B_i n)$$

$$d_{0t'} = \pm 6A_i$$

The overflow restriction for the RiseRun parametric case is:

$$8D_i n^3 w \le 2^{bits-1}.$$

If $bits = 32$, $w = 1280$, and $|D_i| \le 64$ we have

$$10D_i n^3 2^{16} \le 2^{31}$$

$$n^3 < 2^{15}/10,$$

giving $n \le 14$.

If $D_i = 1$, $bits = 32$ and $w = 1280$, we have

$$10n^3 2^{10} \le 2^{32}$$

$$n^3 < 2^{22}/10,$$

giving $n \le 74$.

9. Algorithm Comparison And Evaluation

In this section, we compare the RunRise and RiseRun algorithms as they would be used with parametric curves with the algorithms A and B presented by Klassen in [11]. Table 2 shows the operation costs and the overflow restrictions associated with Klassen's algorithms and our algorithms. We have assumed that a barrel shifter is available. Furthermore, while we generally follow Klassen's practice of weighing each branch of a conditional statement equally, we have made an exception for the branches in our algorithms' main loops. In particular, since the RunRise algorithm plots curve segments with absolute slope less than one, we have assumed (rather conservatively) that the algorithm will make 3 X steps for every 2 Y steps, and thus that the `if` statment in figure 7 will be true only 40% of the time. By similar logic, we assume that the `if` statment in figure 9 will be true only 40% of the time. As curve slope nears zero or infinity, these percentages decrease, and algorithm performance improves.

Clearly, the main loops of both the RunRise and RiseRun algorithms use less operations than Klassen's algorithm B, and are comparable in cost to Klassen's algorithm A. During initialization, our algorithms use no expensive divide operations, and use about half the number of add and multiply operations used by Klassen's algorithms. It should be noted, however, that while our algorithms require that a cubic curve be split into segments, both of Klassen's algorithms A and B do not depend on slope. Thus, initialization for the RunRise and RiseRun algorithms will in practice require slightly more addition and multiply operations than Klassen's algorithms, as well as several floating point calculations.

Klassen's algorithm A has by far the most liberal overflow restriction -- it is linear in n, the number of parametric steps, and w, screen width in pixels. The RunRise algorithm and Klassen's algorithm B both have restrictions that are cubic in n, with the RunRise algorithm allowing twice as many parametric steps as algorithm B. The RiseRun overflow restriction, however, is cubic in n and linear in w. This severely restricts the RiseRun algorithm's utility if only 32 bits are available.
It should be noted that the overflow restriction shown for Klassen's algorithms guarantee that *both* initialization and looping will be accomplished without overflow. The restrictions shown for the RunRise and RiseRun algorithms guarantee only that looping will be accomplished without overflow. Initialization of our algorithms requires many more bits for representation than does initialization for Klassen's algorithms (with the exception of algorithm A's extended precision divide (**xdiv**) operation).

Klassen's algorithm A uses a fixed point approach, and thus incorporates an inherent level of error not present in the other algorithms. Both of Klassen's algorithms can be used with rational coefficients, but doing so would require floating point calculation, increase error in algorithm A, and introduce error into algorithm B. Our algorithms remain perfectly accurate even with rational coefficients.

Table 2. A comparison of Klassen's algorithms from [11] with the algorithms presented in this paper as used for plotting parametric curves with integer coefficients. *n* is the number of parametric steps made, *w* is the width of the screen in pixels, *bits* is the number of bits used to represent algorithm values, Z is the number of bits of fractional precision.

Algorithm	Operation Cost											Overflow Restriction
	Main Loop				Initialization							
	+	if	<<	=	+	*	div	xdiv	<<	if	=	
RunRise	3	1	0	3	16	5	0	0	11	2	16	$n^3 < 2^{bits-1}$
RiseRun	2	1	0	2	22	5	0	0	16	2	14	$8wn^3 < 2^{bits-1}$
Klassen's A	3	0	3	3	17+3Z	12	1	5	2	Z	28+2Z	$46wn < 2^{bits-1}$
Klassen's B	10	3	0	10	40	12	5	0	0	6	33	$2n^3 < 2^{bits-1}$

If non-parametric curves are being plotted, overflow restrictions for our algorithms improve: the RunRise algorithm requires only that the product of the rational denominators D_i be less than 2^{bits-1}, and the RiseRun algorithm is similar, but is linear in screen width. Overflow restrictions for Klassen's algorithms in such a case will not show such a significant improvement.

In summary, Klassen's algorithms make efficient use of available bits, but at the price of algorithm speed or accuracy. Our algorithms require more representational bits, but are faster and more accurate. We believe that if word size is 64 or larger, or non-parametric curves are being rendered, our algorithms are clear winners.

10. Conclusions And Future Work

We have presented integer-only algorithms that allow fast, accurate plotting of cubic curves. Analysis shows that using these algorithms to plot parametric curves may require more representational bits than already existing algorithms. But if such bits are not at a premium, or non-parametric curves are being plotted, our algorithms are the algorithms of choice.

We plan to explore further the use of these algorithms with parametric curves, spline curves, and Bezier curves. In particular, we would like to explore the use of these algorithms with adaptive forward differencing.

11. References

1. Bresenham, J. E. "Algorithm for computer control of a digital plotter," IBM Syst. J. 4(1) (1965): 25-30.
2. Bresenham, J. E. "Algorithms for circular arc generation," Fundamental Algorithms for Computer Graphics, R.A. Earnshaw, ed., NATO ASI Series, Vol. F17, Springer Verlag, Berlin, (1985): 197-218.
3. Bresenham, J. E. "A linear algorithm for incremental digital display of digital arcs," Commun. ACM. 20(2) (Feb. 1977): 100-106.
4. Bresenham, J. E. "Run length slice algorithm for incremental lines," Fundamental Algorithms for Computer Graphics, R.A. Earnshaw, ed., NATO ASI Series, Vol. F17, Springer Verlag, Berlin, (1985): 59-104.
5. Chang, S-L., Shantz, M., and Rocchetti, R. "Rendering cubic curves and surfaces with integer adaptive forward differencing," Comput. Graph., 23(3) (Jul. 1989): 157-166.
6. Foley, J., van Dam, A., Feiner, S., and Hughes, J. Computer Graphics: Principles And Practice, Addison-Wesley, Reading, Mass., (1990).
7. Horn, B. K. P. "Circle generators for display devices," Comput. Graph. Image Process. 5 (1976): 280-288.
8. Lien, S-L., Shantz, M., and Pratt, V. "Adaptive forward differencing for rendering curves and surfaces," Comput. Graph., 21(4) (Jul. 1987): 111-118.
9. Kappel, M. R. "An ellipse-drawing algorithm for raster displays," Fundamental Algorithms for Computer Graphics, R.A. Earnshaw, ed., NATO ASI Series, Vol. F17, Springer Verlag, Berlin, (1985): 257-280.
10. Klassen, R. V. "Drawing antialiased cubic spline curves," ACM Trans. Graph. 10(1) (Jan. 1991): 92-108.
11. Klassen, R. V. "Integer forward differencing of cubic polynomials," ACM Trans. Graph. 10(2) (Apr. 1991): 152-181.
12. McIlroy, M. D. "Best approximate circles on integer grids," ACM Trans. Graph. 2(4) (Oct. 1983): 237-263.
13. Metzger, R. A. "Computer generated graphics segments in a raster display," Spring 1969 Joint Computer Journal Conference, AFIPS Conf. Proc.: 161-172.
14. Pitteway, M. "The relationship between Euclid's algorithm and run length encoding," Fundamental Algorithms for Computer Graphics, R.A. Earnshaw, ed., NATO ASI Series, Vol. F17, Springer Verlag, Berlin, (1985): 105-112.
15. Surany, A. P. "An ellipse-drawing algorithm for raster displays," Fundamental Algorithms for Computer Graphics, R.A. Earnshaw, ed., NATO ASI Series, Vol. F17, Springer Verlag, Berlin, (1985): 281-285.
16. Van Aken, J. "An efficient ellipse-drawing algorithm," IEEE Comput. Graph. & Appl. 4(9) (Sept. 1984): 24-35.
17. Van Aken, J., and Novak, Mark. "Curve drawing algorithms for raster displays," ACM Trans. Graph. 4(2) (Apr. 1985): 147-169.
18. Watson, B., and Hodges, L. "A fast algorithm for rendering quadratic curves on raster displays," SEACM Conf. Proc. 27 (Apr. 1989): 160-165.

Parameterization In Finite Precision

Chandrajit L. Bajaj
Department of Computer Sciences
Purdue University
West Lafayette, IN 47907, USA

Andrew V. Royappa
Department of Computer Sciences
Purdue University
West Lafayette, IN 47907, USA

Abstract

Certain classes of algebraic curves and surfaces admit both parametric and implicit representations. Such dual forms are highly useful in geometric modeling since they combine the strengths of the two representations. We consider the problem of computing the rational parameterization of an implicit curve or surface in a finite precision domain. Known algorithms for this problem are based on classical algebraic geometry, and assume exact arithmetic involving algebraic numbers. In this work, we investigate the behaviour of published parametrization algorithms in a finite precision domain and derive succint algebraic and geometric error characterizations. We then indicate numerically robust methods for parameterizing curves and surfaces which yield no error in extended finite precision arithmetic and alternatively, minimize the output error under fixed finite precision calculations.

Keywords: curves and surfaces, geometric modeling, numerical methods, computational algebraic geometry.

Introduction

Algebraic curves and surfaces are the most common representations for curved objects in geometric modeling. Algebraics satisfy *polynomial* equations, usually with rational coefficients. A *rational* algebraic curve or surface is one whose points can be represented as rational functions in some parameters.

Each form has certain benefits and drawbacks. The parametric form is better for rapid display and interactive control; the implicit form defines a half-space naturally and is suited for modeling. The class of all algebraics is also much larger than the class of rational algebraics. Dual forms can have the best of both worlds.

Mathematical techniques from algebraic geometry have recently been applied to the problem of converting between the two forms. While *implicitizing* a parametric curve or surface is always possible, the converse (*rational parameterization*) is not always possible. That is, all algebraics are not rational. However, important classes of curves and surfaces are rational and algorithms for their rational parameterization based on algebraic geometry have been given in [1],[2],[3], [4],[18], and some will be analyzed here. There is also a numerical method due to Jacobi which works by iteratively converting a conic or quadric to standard form (see [11]).

Functionally, rational parameterization takes one implicit equation in n variables, and for each implicit variable returns a rational function in $n-1$ parameters. Since the rational functions have a common denominator, the output can be viewed as consisting of $n+1$ polynomials.

While the input implicit equation is assumed to have rational coefficients, the output polynomials may require algebraic number coefficients, which are (informally speaking) roots of polynomials, such as $\sqrt{2}$. The algorithms based on algebraic geometry assume exact computations. While techniques exist for manipulating algebraic numbers exactly, they are expensive. In this work, we consider parameterization algorithms in a finite precision domain.

This paper is organized as follows. We choose a finite precision numerical domain and explain our general approach to rederiving a parameterization algorithm to work in this domain. First, we analyze algorithms for conics and quadrics, and then analyze an algorithm for singular cubic curves. The error in each algorithm is described algebraically. We then use the algebraic error analysis to derive simple geometric error bounds for conics and quadrics. Finally, we consider singular cubic parameterization from another standpoint, showing that they can in fact be parameterized exactly using only rational arithmetic. Finally, we conclude by briefly discussing extensions of this approach, e.g. to cubic surfaces.

Approach and Numerical Model

To examine this problem when exact arithmetic is not allowed, we focus on the use of algebraic numbers. We stop short of allowing floating-point arithmetic; instead, the algorithms will use rational arithmetic throughout. Algebraic numbers will be approximated by rationals. This allows us, as a first study, to isolate the effects of the error caused by rational approximations to algebraic numbers.

Recall that parameterization algorithms take a polynomial with rational coefficients as input, and output several polynomials. The algorithms can be restructured so that each coefficient of an output polynomial is given as a *formula* in the (symbolic) input coefficients, and some additional symbols. Every algebraic number required by the parameterization will be represented by one symbol in the formulas.

If the algebraic numbers themselves are substituted for their symbols into the output formula, the output will be exact. However, we only allow rational approximations to algebraic numbers. Substituting these numbers will yield only an approximate output. This output will converge to the exact one as the rational approximations converge to the algebraic numbers.

Thus, given a certain precision to which algebraic numbers are to be rationally approximated, and a bound on the size of the rational input coefficients, one can calculate from the formulas a bound on the rational output coefficients, and hence finally a bound on the precision required to carry out the entire calculation, if fixed finite precision is desired.

Our approach to restructuring the algorithms consists of examining them step by step, and eliminating from each step every subexpression that must vanish if exact arithmetic was used. As it turns out, there are two benefits of this approach: it is often possible to carry through the computation so that the output is expressed as a formula in the input, and error formulas are easily derived. While this method seems to work very well for parameterization algorithms, its applicability in other settings is likely to be limited, where *repeated* computations with algebraic numbers may be required.

Conic Parameterization

We restructure the algorithm in [1] for conic parameterization. The algorithm is given for conics in homogeneous form; this allows the use of both projective and affine transformations. The algorithm is then analyzed for the error in its output when approximations are used for algebraic numbers.

Given the equation of a conic plane curve, parameter functions for the curve are derived. The parameter functions are given as closed form formulas in the parameter t, the coefficients of the curve, and the coordinates of a point on the curve.

INPUT. An irreducible conic curve given by $f(x,y) = a_{20}y^2 + a_{11}xy + a_{02}x^2 + a_{10}y + a_{01}x + a_{00} = 0$.

OUTPUT. Rational functions $(x(t), y(t))$ of degree at most two, such that $f(x(t), y(t)) = 0$.

ALGORITHM.

1. Homogenize the conic. This yields the homogeneous equation $F(X,Y,W) = a_{20}Y^2 + a_{11}XY + a_{02}X^2 + a_{10}YW + a_{01}XW + a_{00}W^2 = 0$. If the X^2, Y^2 or W^2 term is missing from the conic's equation, then it will be linear in the corresponding variable, and can be immediately parameterized. Compute quadratic polynomials $X(t), Y(t)$ and $W(t)$ such that $F(X,Y,W) = 0$, and go to step 4.

2. If all squared terms are present, apply a linear transformation to cancel one of these terms.

3. Parameterize the transformed conic, and apply the inverse transformation to the parameterization; yielding three quadratic polynomials $X(t), Y(t)$ and $W(t)$ such that $F(X,Y,W) = 0$.

4. The parameterization for the affine conic is then given by $x(t) = X(t)/W(t), y(t) = Y(t)/W(t)$.

TRANSFORMATIONS. If all three squared terms are present, then any one of the following three transformations may be used in step 2 of the conic parameterization algorithm. The transformations to cancel X^2 and Y^2 are more general than that for W^2. Hence we explain the X^2 case in most detail (the Y^2 case is similar and omitted).

• To cancel the X^2 term, use the transformation

$$\begin{array}{rcllll} X &=& bX_1 & & & \\ Y &=& cX_1 &+& Y_1 & \\ W &=& dX_1 & & &+\; W_1 \end{array} \tag{1}$$

where (b,c,d) are the homogeneous coordinates of some point on the curve. For the transformation to be well-defined, b must be non-zero. Then, if $d \neq 0$, the transformation is affine; otherwise it is projective. Since proportional projective coordinates represent the same point, we make the restriction $d = 0$ or $d = 1$. If $d = 0$, then we should also make a restriction $b = 1$ or $c = 1$; since $b \neq 0$ is required for the transformation to be well-defined, we will restrict $b = 1$ in this case.

Transforming F yields a new conic curve with implicit equation

$$\begin{aligned} F_1(X_1,Y_1,W_1) &= F(bX_1, cX_1+Y_1, dX_1+W_1) \\ &= F(b,c,d)X_1^2 + F_2(X_1,Y_1,W_1) \end{aligned}$$

Since the subexpression $F(b,c,d)X_1^2$ must vanish, we only need to parameterize

$$\begin{aligned} F_2(X_1,Y_1,W_1) &= (a_{10}d + 2a_{20}c + a_{11}b)X_1Y_1 + \\ &\quad (2a_{00}d + a_{10}c + a_{01}b)X_1W_1 + \\ &\quad a_{20}Y_1^2 + a_{10}Y_1W_1 + a_{00}W_1^2 = 0 \end{aligned}$$

The curve $F_2 = 0$ passes exactly through the point $(1,0,0)$ and can be parameterized by intersecting it with the pencil of lines $Y_1 = tW_1$ which pass through this point, yielding

$$\begin{aligned} X_1(t) &= a_{20}t^2 + a_{10}t + a_{00} \\ Y_1(t) &= -(a_{10}d + a_{11}b + 2a_{20}c)t^2 - \\ &\quad (2a_{00}d + a_{01}b + a_{10}c)t \\ W_1(t) &= -(a_{10}d + a_{11}b + 2a_{20}c)t - \\ &\quad (2a_{00}d + a_{01}b + a_{10}c) \end{aligned} \tag{2}$$

This symbolic parameterization for F_2 is independent of the specific values for b, c and d, i.e., it is always exact, since only rational operations in the coefficients of F_2 are used.

Since $F(b,c,d) = 0$, $F_1(X_1,Y_1,W_1) = F_2(X_1,Y_1,W_1)$, and hence the parameterization (2) also applies to F_1. Applying the inverse linear transformation to this parameterization immediately yields a formula for the original conic:

$$\begin{aligned} X(t) &= b(a_{20}t^2 + a_{10}t + a_{00}) \\ Y(t) &= -(a_{10}d + a_{11}b + a_{20}c)t^2 - \\ &\quad (2a_{00}d + a_{01}b)t + a_{00}c \\ W(t) &= a_{20}dt^2 - (a_{11}b + 2a_{20}c)t - \\ &\quad (a_{00}d + a_{01}b + a_{10}c) \end{aligned} \tag{3}$$

• The transformation cancelling the W^2 term is always affine (i.e. $d = 1$); it is the translation taking the point $(b, c, 1)$ to the affine origin $(0, 0, 1)$. The parameterization formulas derived are

$$\begin{aligned} X(t) &= -(a_{10}d + a_{20}c + a_{11}b)t^2 - \\ &\quad (a_{01}d + 2a_{02}b)t + a_{02}c \\ Y(t) &= a_{20}bt^2 - (a_{10}d + 2a_{20}c)t + (a_{01}d + \\ &\quad a_{11}c + a_{02}b) \\ W(t) &= d(a_{20}t^2 + a_{11}t + a_{02}) \end{aligned} \tag{4}$$

Backward Error Analysis: Conics

The only computation in the algorithm given above is to derive the coordinates of a point on the input conic curve. Once these coordinates are found, the parameterization is given as a closed form formula in terms of those numbers and the coefficients of the input curve. The output parameter functions $x(t)$ and $y(t)$ are formulas in algebraic numbers b and c (d is always either 0 or 1), satisfying $f(x(t), y(t)) = 0$. When approximations $\tilde{b}$ and $\tilde{c}$ are used for b and c, the algorithm will output approximate parameter functions $\tilde{x}(t)$ and $\tilde{y}(t)$ such that $f(\tilde{x}(t), \tilde{y}(t)) \neq 0$. These parameter functions also correspond to some algebraic curve. We would like to find the implicit equation of this new curve and compare it to the original input curve. This is the approach of backward error analysis.

LEMMA. Let the first transformation above be used in computing the parameterization. Then the output parametric curve exactly satisfies the perturbed implicit equation $\tilde{f}(x, y) = a_{20}y^2 + a_{11}xy + (a_{02} - \delta)x^2 + a_{10}y + a_{01}x + a_{00} = 0$, where the value δ is given by

$$\delta = \begin{cases} \dfrac{f(\tilde{b}, \tilde{c})}{\tilde{b}^2} & \text{if } d = 1 \\ a_{20}\tilde{c}^2 + a_{11}\tilde{c} + a_{02} & \text{if } d = 0 \end{cases}$$

PROOF. The analysis begins by computing the value of the expression $f(\tilde{x}(t), \tilde{y}(t))$. This value must vanish when exact arithmetic is used, since every point on the output (parametric) curve must be on the input (implicit) curve. However, in the presence of numerical approximations, it will be nonzero, and can be found symbolically. It depends on which transformations above was used. In the following, we use the relationship $f(X/W, Y/W) = F(X, Y, W)/W^2$. We now compute $f(\tilde{x}(t), \tilde{y}(t))$ directly:

$$\begin{aligned} f(\tilde{x}(t), \tilde{y}(t)) &= f(\frac{\tilde{X}(t)}{\tilde{W}(t)}, \frac{\tilde{Y}(t)}{\tilde{W}(t)}) \\ &= \frac{F(\tilde{X}(t), \tilde{Y}(t), \tilde{W}(t))}{\tilde{W}^2(t)} \\ &= \frac{F(\tilde{b}\tilde{X}_1(t), \tilde{c}\tilde{X}_1(t) + \tilde{Y}_1(t), d\tilde{X}_1(t) + \tilde{W}_1(t))}{\tilde{W}^2(t)} \\ &= \frac{F_1(\tilde{X}_1(t), \tilde{Y}_1(t), \tilde{W}_1(t))}{\tilde{W}^2(t)} \\ &= \frac{F(\tilde{b}, \tilde{c}, d)\tilde{X}_1^2(t) + F_2(\tilde{X}_1(t), \tilde{Y}_1(t), \tilde{W}_1(t))}{\tilde{W}^2(t)} \\ &= \frac{F(\tilde{b}, \tilde{c}, d)\tilde{X}_1^2(t)}{\tilde{W}^2(t)} \\ &= \frac{F(\tilde{b}, \tilde{c}, d)}{\tilde{b}^2}\frac{\tilde{X}^2(t)}{\tilde{W}^2(t)} \\ &= \frac{F(\tilde{b}, \tilde{c}, d)}{\tilde{b}^2}\tilde{x}^2(t) \end{aligned}$$

The key is that $F_2(\tilde{X}_1(t), \tilde{Y}_1(t), \tilde{W}_1(t)) = 0$ even when approximations are used.

Thus each point on the output curve evidently satisfies the equation $f(x, y) - (F(\tilde{b}, \tilde{c}, d)/\tilde{b}^2)x^2 = 0$. Since $F(\tilde{b}, \tilde{c}, 1) = f(\tilde{b}, \tilde{c})$ and $F(1, \tilde{c}, 0) = a_{20}\tilde{c}^2 + a_{11}\tilde{c} + a_{02}$, the lemma follows. □

Similarly, one can show that if the second transformation is used, the approximate output parameterization satisfies the equation $f(x, y) - \delta y^2 = 0$ with $\delta = f(\tilde{b}, \tilde{c})/\tilde{c}^2$, for $d = 1$, and $\delta = a_{20} + a_{11}\tilde{b} + a_{02}\tilde{b}^2$ for $d = 0$.

Finally, if the third transformation was used, then the output parametric curve satisfies the implicit equation $f(x, y) - \delta = 0$, where $\delta = f(\tilde{b}, \tilde{c})$.

Thus the effect of approximating (b, c) by rationals is an output parametric curve that corresponds to the input implicit curve, perturbed in precisely one of the coefficients a_{02}, a_{20}, a_{00}, depending on the transformation used.

We note that the above discussion remains valid under scaling of the input equation by a constant.

Geometric Error Bounds: Conics

The algebraic error analysis tells us the implicit equation of the approximate output curve; it is natural to investigate the relationship between the input (exact) and output (approximate) curves. For conics and quadrics, we can derive geometric error bounds in terms of the magnitude of the coefficient perturbation.

In [10], general bounds are given for local geometric perturbations at a point on a curve due to random perturbations in the coefficients of its equation. However, the perturbations that appear as a result of approximations in the parameterization process have a definite structure, which we exploit to derive global geometric error bounds.

We investigate the geometric effects of perturbing a single coefficient in the equation of a conic curve. The perturbations

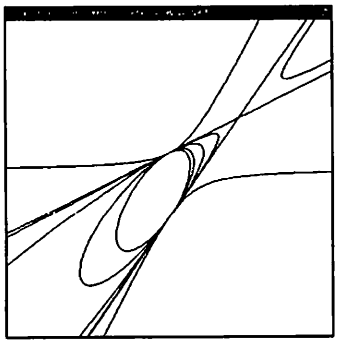


Figure 1: Conics Perturbed in the Higher Order Coefficients

yield an entire family of conics. In particular, the effect of perturbing the constant coefficient is investigated.

It will be shown that perturbing the constant coefficient gives rise to a conic similar to the original conic. We then bound the maximum orthogonal distance between the original and perturbed conic.

We first list some relevant facts about conics, from [19] and [16]. Consider the affine quadratic equation of a conic curve C, in the form

$$F(x,y) = ax^2 + by^2 + 2hxy + 2gx + 2fy + c = 0$$

The *discriminant* of C is

$$\Delta = \begin{vmatrix} a & h & g \\ h & b & f \\ g & f & c \end{vmatrix} = abc + 2fgh - af^2 - bg^2 - ch^2$$

The following facts about conics are known:

1. C degenerates to a pair of lines when $\Delta = 0$
2. C is a parabola when $ab - h^2 = 0$, an ellipse when $ab - h^2 < 0$, and a hyperbola when $ab - h^2 > 0$.
3. When C is not a parabola, its center is given by $(\frac{hf-bg}{ab-h^2}, \frac{gh-af}{ab-h^2})$.
4. The axes of the conic are given by the equation $h(x^2 - y^2) - (a-b)xy = 0$.
5. The conic can be translated to have its center at the origin, and axes rotated to the principal axes. In this coordinate system its equation is

$$F(x,y) = (a+b+R)x^2 + (a+b-R)y^2 + \frac{2\Delta}{ab-h^2} = 0$$

where $R^2 = (a-b)^2 + 4h^2$. It is clear that perturbing the constant term c in the equation of a conic will produce a new conic of the same type that is concentric and coaxial with the original (see also [7]). Perturbing the coefficients of x^2 or y^2, on the other hand, can change all these quantities: Figure 1 shows a family of conics perturbed only in the coefficient of x^2; they vary in type, center, and axis. We will therefore only consider the third transformation of the conic algorithm, which only perturbs the constant coefficient.

Even when only the constant coefficient is perturbed, the conic could still degenerate into a pair of lines. A large enough perturbation could turn a hyperbola into one that is concentric and coaxial to the original, but with transverse and conjugate axes reversed. Hence, an upper bound must be imposed on the perturbation. Since the constant coefficient c appears linearly in the discriminant Δ, so will the perturbed coefficient $c + \delta$, and hence one can immediately bound $|\delta|$ to avoid this case. If this bound is very small the conic will already be close to degenerate.

For perturbations smaller than this bound, then, we wish to geometrically describe the error. Define the (orthogonal) distance from a point p on one conic to the other conic as the shortest distance along the normal vector at p to the other conic. Then the maximum orthogonal distance from a point on one conic to the other will occur at one of the extreme points of the conic along its semi-axes, if ellipse, or transverse axis, if hyperbola.

Now suppose one is given two conics C, $\tilde{\text{C}}$, where the second conic is derived by perturbing the constant coefficient in the equation of the first (if C is a parabola, some slight modifications will apply to the arguments below). Then they will be concentric and coaxial, and we can consider their equations in a coordinate system where their center is at the origin and their axes are aligned with the primary axes. In this coordinate system their equations will take the form $f(x,y) = Ax^2 + By^2 + C_1 = 0$ and $\tilde{f}(x,y) = Ax^2 + By^2 + C_2 = 0$. Let $d_x, \tilde{d}_x$ be the distances along the x-axis from the origin (which is the center) to C, $\tilde{\text{C}}$ respectively. Likewise, let $d_y, \tilde{d}_y$ be the distances along the y-axis. That is, d_x an d_y are simply the lengths of the semi-axes of the conic (in the case of a hyperbola, only one of these distances is finite). Then $\tilde{d}_x = d_x + p_x$ and $\tilde{d}_y = d_y + p_y$. One of p_x and p_y will be the maximum orthogonal distance between the two curves. We can solve directly for p_x and p_y, the maximum and minimum geometric error.

To solve for p_x, put $y = 0$ in the curve equations. Then $d_x^2 = -\frac{C_1}{A}$ and $\tilde{d}_x^2 = -\frac{C_2}{A}$. Hence $\tilde{d}_x^2 - d_x^2 = (d_x + p_x)^2 - d_x^2 = \frac{C_1 - C_2}{A}$. So $p_x^2 + 2d_x p_x = \frac{C_1 - C_2}{A}$, and since $p_x = 0$ when $C_1 - C_2 = 0$, we find that $p_x = -d_x + \sqrt{d_x^2 + \frac{(C_1 - C_2)}{A}}$. Revert to the original coordinate system, where the conics have equations $ax^2 + by^2 + 2hxy + 2gx + 2fy + c_1 = 0$, $ax^2 + by^2 + 2hxy + 2gx + 2fy + c_2 = 0$; then, by the coordinate transformations of the previous section, and some algebra, $C_1 - C_2 = 2(c_1 - c_2)$. Using the definitions for A and B, and R as given in previously, and putting $\delta = c_1 - c_2$, we find that

$$p_x = -d_x + \sqrt{d_x^2 + \left(\frac{2}{a+b+R}\right)\delta}$$

Now suppose it is desired that $|p_x| < \epsilon$ for some $\epsilon > 0$, and we wish to bound $|\delta|$. To have $|p_x| < \epsilon$, it is necessary that $p_x < \epsilon$ and $p_x > -\epsilon$. For reasons that will soon be clear, we

will require $\epsilon < d_x$. Considering each case separately,
1. $p_x < \epsilon$ implies that $\left(-d_x + \sqrt{d_x^2 + \left(\frac{2}{a+b+R}\right)\delta}\right) < \epsilon$.
2. $p_x > -\epsilon$ implies $\left(-d_x + \sqrt{d_x^2 + \left(\frac{2}{a+b+R}\right)\delta}\right) > -\epsilon$.
After considering both possibilities for the sign of $a + b + R$, the requirements above may be satisfied by taking

$$\begin{aligned} |\delta| &< \epsilon(2d_x + \epsilon)\,|(a+b+R)/2| \\ |\delta| &< \epsilon(2d_x - \epsilon)\,|(a+b+R)/2| \end{aligned}$$

for cases (1) and (2) respectively.
Finally, recalling that $\epsilon < d_x$, choices (1) and (2) can be simultaneously satisfied by choosing

$$|\delta| < \epsilon \cdot d_x \cdot \left|\frac{a+b+R}{2}\right|$$

This is the only simplification made in the calculation, and at most a factor of two of accuracy (one bit) is lost.
The error along the y axis is bounded in an identical way. The quantities d_x and d_y are independent of any scaling of the coefficients of the original conic by a constant, but the scale factor will be linearly present in the quantities $a + b + R$ and $a + b - R$. Hence, if δ is defined as in the backward error analysis for conics, these bounds correct for the scale factor automatically. Keeping this in mind, it suffices to compute δ such that

$$|\delta| < \epsilon \cdot \frac{\min(d_x \cdot |a+b+R|, d_y \cdot |a+b-R|)}{2}$$

Quadrics

The results for conics generalize directly to quadrics. It is possible to derive explicit formulas of degree two for the parameterization, in a pair of parameters s, t. The formulas are small; the only computation required is that of finding a point (a, b, c, d) on the homogeneous conic. There are four choices of transformations, one to cancel each squared term. A corresponding error analysis holds. For instance, if the W^2 term is cancelled using an approximate (affine) point $(\tilde{a}, \tilde{b}, \tilde{c})$, then the output parameterization will satisfy $f(\tilde{x}(s,t), \tilde{y}(s,t), \tilde{z}(s,t)) - f(\tilde{a}, \tilde{b}, \tilde{c}) = 0$, i.e., the original input equation perturbed in the constant coefficient.
This raises an important point. In general, a parametric curve of degree n corresponds to a curve of algebraic (implicit) degree n, but a parametric surface of degree n may correspond to a surface of algebraic degree up to n^2. Thus when using approximations in a parameterization, one might legitimately question whether the algebraic degree of the output is the same as that of the input. In the case of quadrics ($n = 2$), it would be unpleasant if a parameterization algorithm could actually output a cubic or quartic surface. Fortunately, the error analysis above allows us to answer this question in the negative, since a set of rational parametric equations of a surface satisfy a unique irreducible algebraic surface.

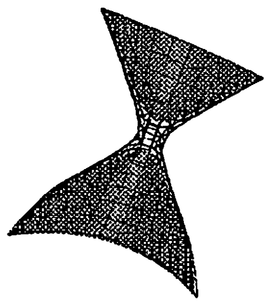


Figure 2: Quadrics Perturbed in the Constant Coefficient

Geometric Error Bounds: Quadrics

As for conics, there is a quadric discriminant. The sign of the discriminant, among other quantities, distinguishes amongst the various quadric surfaces. Essentially, perturbing the constant coefficient preserves the center and orientation, although the quadric could degenerate from a hyperboloid of one sheet to a cone to a double-sheeted hyperboloid (see Figure 2). Perturbing the highest order coefficients could cause an ellipsoid to change to a cylinder to a one-sheeted hyperboloid, for example, in addition to changing its orientation and center (Figure 3). Since the geometric errors find their extrema along the axes when the center and orientation are fixed, we can bound the errors easily in this case. We simply state the results, for brevity. Vital information regarding quadrics was taken from [20].
Let two quadrics that differ only in their constant coefficient be given. Generalizing the notation from the conic case, let d_x, d_y, d_z be the distances from the origin to the unperturbed conic (some may not be finite). Given a number $\epsilon > 0$ that also satisfies $\epsilon < \min(d_x, d_y, d_z)$, and a difference in the constant coefficients of a quantity δ, if the geometric perturbations p_x, p_y, p_z are to satisfy

$$\max(|p_x|, |p_y|, |p_z|) \quad < \quad \epsilon$$

then it suffices to choose δ such that

$$|\delta| < \epsilon \cdot \min(d_x \cdot |\lambda_1|, d_y \cdot |\lambda_2|, d_z \cdot |\lambda_3|)$$

where expressions for λ_i are the roots of a cubic polynomial $\phi(\lambda)$ whose coefficients are expressions in the coefficients of the quadrics. From data in [20], the quadric can be put in standard form in terms of the roots of $\phi(\lambda)$, allowing the the quantities d_x, d_y, d_z to be efficiently calculated. We omit the details here.

Only considering real values of d_x, d_y, d_z, then, we can bound the geometric error for a quadric due to approximate parameterization.

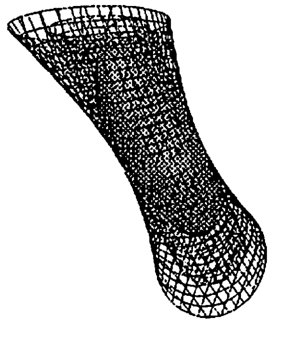


Figure 3: Quadrics Perturbed in the Higher Order Coefficients

Singular Cubic Curves

For an irreducible, singular cubic plane curve, a parameterization algorithm is given in [2], which we analyze. While output formulas exist for this case, they are unwieldy, and instead we show how they can be derived, and the error in the parameterization.

INPUT. A cubic plane curve given by the cubic equation $f(x,y) = a_{30}y^3 + a_{21}xy^2 + a_{20}y^2 + a_{12}x^2y + a_{11}xy + a_{10}y + a_{03}x^3 + a_{02}x^2 + a_{01}x + a_{00} = 0$.

OUTPUT Rational functions $(x(t), y(t))$ of degree at most four, such that $f(x(t), y(t)) = 0$.

ALGORITHM. As in the conic case, the curve is transformed into a birationally equivalent one that is readily parameterizable. Several transformations are used. The steps are detailed below. If the cubic has a zero x^3 or y^3 term, the first step is omitted, otherwise the first step cancels y^3. The computation is symmetric with respect to x.

1. Apply a transformation that removes the y^3 term of f. This can be done via the linear transformation $x = x_1 + qy_1, y = y_1$. When applied to the cubic equation $f(x,y) = 0$, this yields a new cubic curve with equation $f_1(x_1, y_1) = 0 = f(x_1 + qy_1, y_1) = L(q)y_1^3 + f_2(x_1, y_1)$ where $L(q) = a_{03}q^3 + a_{12}q^2 + a_{21}q + a_{30}$. Choose q to be a root of L, i.e. $L(q) = 0$. Then the subexpression $L(q)y_1^3$ must vanish, so we only need to parameterize the curve $f_2(x_1, y_1) = 0$

2. Parameterize the cubic with equation $f_2(x_1, y_1) = 0$, which has no y_1^3 term, by transforming it into a transformed into a quadratic curve. f_2 is of the form

$$f_2(x_1, y_1) = g_1(x_1)y_1^2 + g_2(x_1)y_1 + g_3(x_1) \qquad (5)$$

where g_1, g_2, g_3 have degrees equal to their subscripts. The *discriminant* of f_2 (with respect to y_1) is simply $g_4(x_1) = g_2(x_1)^2 - 4g_1(x_1)g_3(x_1)$. It can be shown that $g_4(x_1)$ must have a multiple root of the original cubic is singular, as assumed. By performing the following substitution

$$y_2 = 2g_1y_1 + g_2 \qquad (6)$$

we have

$$\begin{aligned} 4g_1f_2 &= 4g_1^2y_1^2 + 4g_1g_2y_1 + 4g_1g_3 \\ &= (2g_1y_1 + g_2)^2 - (g_2^2 - 4g_1g_3) \\ &= y_2^2 - g_4 \end{aligned} \qquad (7)$$

Note that $g_4(x_1)$ is a polynomial in x_1 of degree at most four. The curve is singular (and hence rational) if and only if $g_4(x_1)$ has a multiple root. This repeated root can be real or complex; only the real case is considered. Now for any number r, expand the polynomial $g_4(x_1)$ in a Taylor series at r: $g_4(x_1) = \sum_{i=0}^{4} \frac{g_4^{(i)}(r)}{i!}(x_1 - r)^i$. The terms of order higher than 4 are identically zero, g_4 being a polynomial of degree 4. Collecting coefficients of $(x_1 - r)^2$ yields

$$g_4(x_1) = q_2(x_1)(x_1 - r)^2 + g_4'(r)(x_1 - r) + g_4(r) \qquad (8)$$

where $q_2(x_1)$ is of degree two. Now apply the substitution $y_3 = y_2/(x_1 - r)$ together with (8) into the right-hand side of (5); this leads to

$$\begin{aligned} &4g_1f_1 = y_2^2 - g_4(x_1) \\ &= (y_3^2 - q_2(x_1))(x_1 - r)^2 + g_4'(r)(x_1 - r) + g_4(r) \qquad (9) \\ &= f_3(x_1, y_3) \end{aligned}$$

Choose r to be a multiple root of $g_4(x_1)$: then $g_4(r) = g_4'(r) = 0$, and the subexpression $g_4'(r)(x_1 - r) + g_4(r)$ must vanish. Therefore, to parameterize $f_3(x_1, y_3)$ we can simply parameterize the conic curve corresponding to the quadratic factor $C(x_1, y_3) = y_3^2 - q_2(x_1) = 0$.

3. Parameterize the conic with equation $C(x_1, y_3) = 0$ using the methods of the previous section. This yields a pair of rational functions $(x_1(t), y_3(t))$ that satisfy $C(x_1(t), y_3(t)) = 0$. Applying all the transformations in reverse yields one for the input cubic.

The cubic parameterization calls for computing a root q of the cubic polynomial $L(q)$, a multiple root r of the quartic polynomial $g_4(x_1)$, and a parameterization $(x_1(t), y_3(t))$ of the conic with equation $C(x_1, y_3) = 0$. Assuming, say, that the third conic transformation section was used, a pair of algebraic numbers (b, c) need to be computed.

Backward Error Analysis: Singular Cubics

If all computations were exact, i.e. $L(q) = 0$, $g_4(r) = 0$, and $C(x_1(t), y_3(t)) = 0$, then the output will be correct. However, one may need to use approximations $\tilde{q}$, $\tilde{r}$ and $(\tilde{b}, \tilde{c})$, which will lead to an approximate output parameterization $(\tilde{x}(t), \tilde{y}(t))$. In this case one must measure the error incurred. Once again, a backward error analysis will be performed, beginning with back-substitution.

LEMMA. The output parameterization will satisfy the implicit equation

$$f(x,y) - L(\tilde{q})y^3 - \frac{C(\tilde{b},\tilde{c})(x - \tilde{q}y - \tilde{r})^2 + g_4'(\tilde{r})(x - \tilde{q}y - \tilde{r}) + g_4(\tilde{r})}{4g_1(x - \tilde{q}y)} = 0$$

PROOF. Given the approximate output parameter functions $(\tilde{x}(t), \tilde{y}(t))$, we compute $f(\tilde{x}(t), \tilde{y}(t))$. The subscript (t) is dropped for convenience. Then $f(\tilde{x}, \tilde{y}) = f(x_1 + \tilde{q}y_1, y_1) = L(\tilde{q})y_1^3 + f_2(x_1, y_1)$.
Applying transformations in reverse and performing some algebraic manipulation, we find that

$$f_2(\tilde{x_1}, \tilde{y_1}) = f_2(\tilde{x_1}, \frac{\tilde{y_2} - g_2(\tilde{x_1})}{2g_1(\tilde{x_1})}) = \frac{\tilde{y_2}^2 - g_4(\tilde{x_1})}{4g_1(\tilde{x_1})}$$
$$= \frac{(\tilde{y_3}^2 - q_2(\tilde{x_1}))(\tilde{x_1} - \tilde{r})^2 + g_4'(\tilde{r})(\tilde{x_1} - \tilde{r}) + g_4(\tilde{r})}{4g_1(\tilde{x_1})}$$

Now $C(\tilde{x_1}(t), \tilde{y_3}(t)) = \tilde{y_3}^2 - q_2(\tilde{x_1})$, and since we assumed that the third conic transformation was used to parameterize C, it follows that there is a point $(\tilde{b}, \tilde{c})$ such that $C(\tilde{x_1}(t), \tilde{y_3}(t)) = C(\tilde{b}, \tilde{c})$. The lemma follows by substituting and expanding previous identities. □
If the values $\tilde{q}, \tilde{r}, \tilde{b}, \tilde{c}$ are exact, then $L(\tilde{q}) = g_4(\tilde{r}) = g_4'(\tilde{r}) = C(\tilde{b}, \tilde{c}) = 0$, and it is clear that the parametric output curve coincides with the implicit input curve.
However, if the values are *not* exact, the output curve differs from the input curve. The coefficient perturbations are now present in many terms, not just one.

Exact Solutions: Singular Cubics

Finally, we show that in some cases, algebraic number computation is unnecessary for exact rational parameterization. A fact that appears to be known in Diophantine analysis is that a rational cubic curve with rational coefficients has a rational singular point. [1] This was apparently not well-known in the geometric modeling community; it is mentioned in a book on Diophantine equations ([14]).
Every rational cubic has a singular point. It is well-known (see, e.g. [21] for details) that such a cubic can be parameterized by a pencil of lines through the singularity, which then intersect the cubic at exactly one other point. The coordinates of the latter point parameterized by the slope of the line give parameter functions for the cubic curve. The parameter functions are given as closed form formulas in the parameter t, the coefficients of the curve, and the coordinates (b, c) of the singularity, as shown below:

$$\begin{aligned} X(t) &= a_{30}bt^3 - (3a_{30}c + a_{20})t^2 - \\ &\quad (2a_{21}c + a_{12}b + a_{11})t - (2a_{03}b + a12c + a_{02}) \\ Y(t) &= -((2a_{30}c + a_{21}b + a_{20})t^3 + \\ &\quad (a_{21}c + 2a_{12}b + a_{11})t^2 + (3a_{03}b + a_{02})t - a_{03}c) \\ W(t) &= a_{30}t^3 + a_{21}t^2 + a_{12}t + a_{03} \end{aligned}$$

Therefore, if extended precision rational arithmetic is allowed, one can theoretically parameterize an irreducible rational cubic curve without error and without algebraic number computation, by computing the singular point exactly, and substituting the coordinates in the above formula. One way to compute the singularity rationally is as follows.

[1]We are grateful to Allan Adler for alerting us to this fact.

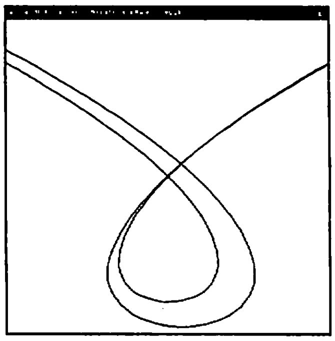


Figure 4: Exact and Perturbed Singular Cubics

An affine singular point is found as a solution to the system of equations $f(x,y) = f_x(x,y) = f_y(x,y) = 0$. The x-coordinate of this solution will be a multiple rational root of the degree six polynomial $p(x) = resultant(f(x,y), f_x(x,y), y)$. The rational roots of a polynomial can be computed by applying the algorithm in ([13]). The resultant is computed using a subresultant remainder sequence; this may then be used to compute the y-coordinate ([4]). Each (x, y) pair found this way can be tested whether it additionally satisfies $f_x(x,y) = 0$; only one pair will satisfy the test.

Extensions

We have reformulated algorithms for rational parameterizations in a finite precision domain. Algebraic numbers are approximated by rationals to produce an approximate parameterization of an implicit curve or surface. For each method, we isolated the error due to the algebraic number approximation. The error formulas have useful geometric interpretations, some examples of which were given. In ongoing research, we find that the parameterization algorithms are quite stable. For instance, monoid parameterizations depend on the computation of the singular point (b, c) of a monoid curve. If a rational approximation $(\tilde{b}, \tilde{c})$ is calculated instead, we show that the monoid algorithm can be formulated so that the approximate output parametric curve will have a singularity at $(\tilde{b}, \tilde{c})$. In general, if a point is computed for a parameterization (conic, quadric, monoidal curve or surface), we show that the approximate point and the approximate output mimic the relationship of the exact point and the (exact) input (see Figure 4).
The method of rederiving the algorithms to work properly in finite precision arithmetic works well for parameterization. In fact, we have been able to generate a formula to parameterize a cubic surface, in terms of its coefficients and certain algebraic numbers derived from them [15]. The rational parametric equations derived are of the fourth degree, comparable to [2], [18]. Our algorithms will always succeed in producing output of the right degree when approximations are used. A straightforward

MACSYMA implementations of [2] fails in this case. This is because such algorithms require rational simplification of multivariate expressions, whereas simple divisibility properties are destroyed by small numerical perturbations, leading to intermediate and output expression swell.
While the general formula is very large, it is easily implemented in common programming languages and we expect it to be highly efficient. Furthermore, by specializing certain quantities in the formula, we hope to derive parameterization formulas for various families of cubic surfaces, and build a menagerie of controllable dual form surfaces.
Finally, we have only considered these problems in a rational arithmetic setting (extended and finite precision). However, having reduced the parameterization algorithms to simple formulas involving only additions and multiplications, we would like to experiment with them in the floating point realm, also.

Acknowledgements. This research was supported in part by NSF grants CCR 90-00028, DMS 91-01424 and AFOSR contract 91-0276.

References

[1] Abhyankar, S. S., and Bajaj, C., (1987a), Automatic Parameterization of Rational Curves and Surfaces I: Conics and Conicoids, *Computer Aided Design*, 19, 1, 11 - 14.

[2] Abhyankar, S. S., and Bajaj, C., (1987b), Automatic Parameterization of Rational Curves and Surfaces II: Cubics and Cubicoids, *Computer Aided Design*, 19, 9, 499 - 502.

[3] Abhyankar, S. S., and Bajaj, C., (1988), Automatic Parameterization of Rational Curves and Surfaces III: Algebraic Plane Curves, *Computer Aided Geometric Design*, 5, 309 - 321.

[4] Abhyankar, S., and Bajaj, C., (1989), "Automatic Rational Parameterization of Curves and Surfaces IV: Algebraic Space Curves", *ACM Transactions on Graphics*, (1989), 8, 4, 325-334.

[5] Bajaj, C., (1990), "Rational Hypersurface Display," *Computer Graphics*, 24, 2, 117-128.

[6] Bajaj, C., and Royappa, A., (1990) "The Ganith Algebraic Geometry Toolkit", in *DISCO '90*, Capri, Italy.

[7] Bookstein, F. L., (1979), "Fitting Conic Sections to Scattered Data," *Computer Graphics and Image Processing*, 9, 56-71.

[8] Canny, J. (1988) "Elimination Theory", *The Complexity of Robot Motion Planning*, ACM Doctoral Dissertation Series, MIT Press, chapter 3.

[9] Collins, G. (1967), "Subresultants and Reduced Polynomial Remainder Sequences," *JACM*, Vol. 14, No. 1, Jan. 1967, pp. 128-142.

[10] Farouki, R.T., (1987) On the Numerical Condition of Algebraic Curves and Surfaces 1. Implicit Equations, IBM Research Report RC 13263 (#59267), 1987.

[11] Golub, G., and van Loan, C. (1983), *Matrix Computations*, Johns Hopkins Press, pp 444-448.

[12] Hodge, W.V.D., and Pedoe, D. (1952), *Methods of Algebraic Geometry*, v.2, p. 69, Cambridge University Press.

[13] Loos, R., (1983) "Computing Rational Zeroes of Integral Polynomials by p-Adic Expansion", *SIAM J. Computing*, Vol. 12, No. 2, May 1983, pp 286-293.

[14] Mordell, L.J., (1969) *Diophantine Equations*, Academic Press.

[15] Royappa, A., (1992) *Symbolic and Numerical Methods in Geometric Modeling*, Computer Science, Purdue University, Ph.D. Thesis, in preparation.

[16] Salmon, G., (1879), *Conic Sections*, Longmans, Green and Co., London.

[17] Schwartz, J. and Sharir, M., (1983), On the Piano Movers Problem: II, General Techniques for Computing Topological Properties of Real Algebraic Manifolds, *Adv. Appl. Math.* Vol. 4, pp. 298–351.

[18] Sederberg, T., and Snively, J., (1987), "Parameterization of Cubic Algebraic Surfaces", *The Mathematics of Surfaces II*, ed. R. Martin, Oxford University Press, pp 299-321.

[19] Sommerville, D.M.Y., (1924), *Analytical Conics*, G. Bell and Sons, London.

[20] Sommerville, D.M.Y., (1951), *Analytical Geometry of Three Dimensions*, G. Bell and Sons, Cambridge University Press.

[21] Walker, R., (1950), *Algebraic Curves*, Springer Verlag 1978, New York

Hyper-Rendering

Jürgen Emhardt
Thomas Strothotte

Interactive Systems Lab
Department of Computer Science
Free University of Berlin
Nestorstraße 8-9, D-1000 Berlin 31, Germany
{emhardt, strothotte} @inf.fu-berlin.de

Abstract

Current systems for the automatic generation of information presentation or the automatic illustration of objects are mainly generation-oriented, i.e. they generate solutions to a user who in turn has only few possibilities to lead a dialogue with the generation system. By contrast, systems which emphasize dialogues tend to be weak in graphical interaction.
In order to build systems which are both generation- and dialogue-oriented, we present a new software architecture for computer graphics applications. We develop the concept of *hyper-rendering* which produces a formal description of the scene as viewed by the user and can either be carried out within a renderer or in a separate program. The output of the hyper-rendering program is an *image description* which consists of information which conventional renderers usually compute but typically throw away. We describe two broad categories of applications of hyper-rendering and a prototypical implementation. Examples of the pictures produced as well as a sample session with an application are included.

Keywords: rendering, image description, photorealistic images, virtual reality, multi-media, interactive systems.

1. Introduction

Today's graphics systems for producing photorealistic images consist of two main software components: a modeler with which a user can define a scene (model objects to be rendered, choose the camera position and perspective, etc.) and which produces as output a scene description; and a renderer, which accepts as input a scene description and produces as output the image. It is a fundamental characteristic of such graphics systems that the output of the renderer is designed exclusively for viewing by a (human) user. While the machine produces the image based on a scene description, no information is stored in general as to what a user can actually see. This lack of information makes it impossible to establish a link between the renderer and an application program, although a main purpose of producing photorealistic images is *communication*.

Indeed, rendering algorithms "waste" a great deal of information which could be gathered and represented explicitly. For example, hidden surface removal algorithms--as their name implies--throw away information about surfaces which are not visible to a user: But if a user wants to know which objects of the scene she or he *cannot* see? Such information pertaining to objective and subjective perception of the picture by the user and its relationship to the modeled scene is not made available systematically.

In this paper we propose that the usability of graphics systems can be enhanced greatly by making information about the rendering process explicitly available to other programs. We refer to this process of deriving such information as *hyper-rendering*. Hyper-rendering produces a formal description of the scene as viewed by the user and can either be carried out within a renderer or, as in our prototypical implementation, in a separate program. The latter approach has the advantage that separate algorithms can be used for rendering and hyper-rendering of the same image, especially when hardware-rendering is used. Furthermore, when an image is to be produced with ray-tracing it may suffice to carry out the hyper-rendering with a fast z-buffer algorithm. Indeed, in certain applications it may even suffice initially just to carry out hyper-rendering and only later to perform the more time-consuming rendering.

This paper is organized as follows: Previous research is surveyed in Chapter 2. In Chapter 3, we present a new software architecture for graphics systems with integrated hyper-rendering. Next, we point out two broad categories of applications of hyper-rendering and discuss the benefits to end-users. In Chapter 4, we survey the implementation techniques of our prototypical hyper-renderer. Chapter 5 gives an example of the capabilities of an application which uses this hyper-renderer and shows a sample dialogue session. Further work is discussed in Chapter 6.

2. Background

Our work on hyper-rendering is intended as a bridge between the generation-oriented systems which present "canonical" solutions to a mainly *passive* user, and the dialogue-oriented systems which rely on an *active* user and which are able to answer questions. A representative of the first category is APT (A Presentation Tool, [Mackinlay, 1986]). The system generates presentations of two dimensional relational information, where the two main criteria for the generation are *expressiveness* and *effectiveness*. Expressiveness criteria determine whether a graphical language can express the desired information, and effectiveness criteria determine whether a graphical language exploits the capabilities of the output medium and the human visual system. Thus, users get a large variety of "canonical" presentations fulfilling these two criteria. However, the techniques developed by Mackinlay are very difficult to extend to user-interaction, for example, for cases in which users wish to modify the presentation. The information about such a modification of the layout is difficult to report back to the generation system, which is in addition not able to judge the quality of the modification.

The WIP system (Knowledge-based Presentation of Information, [Wahlster et al., 1991]) generates a variety of multimodal documents from an input consisting of a formal description of the communicative intent of a planned presentation. The focal point of WIP is the *generation* of illustrated texts that are customized for the intended audience and situation, but interaction with the illustrations is not yet possible.

A representative of generation-oriented dialogue systems is the IBIS system of [Seligmann and Feiner, 1991] (see also [Feiner, 1985]). It is intended for the automatic generation of intent-based illustrations and shares several research interests with the WIP system, but differs in the system's architecture, for example. An *illustration* is a picture that is designed to fulfill a communicative intent such as showing the location of an object or showing how an object is manipulated. The design of an illustration is treated as a goal-driven process within a system of constraints. The system uses a generate-and-test approach which relies on a rule-based system of methods and evaluators. Methods are rules that specify how to accomplish a visual effect, while evaluators are rules that specify how to determine how well a visual effect is attained in an illustration. As users are able to manipulate illustrations interactively, IBIS maintains the methods of *visibility* and *recognizability* during an interactive session. The current implementation supports user-controlled view specification, that is, the user can zoom objects or specify a new camera location. However, there is no possibility for a user to *ask* anything, for example about the interior of a particular object or to find out through a dialogue where shadows are located. However, the visibility methods of the IBIS system do provide very simple information for further processing through application modules. For example, the evaluator which calculates whether an object which must be visible is partially obscured returns a binary value.

More flexibility is desirable for human-computer dialogues concerning the graphics, particularly in user interfaces using graphics for teaching purposes. Flexibility here refers not only to the viewing specification but also to supplementary information about the objects. To this end, [Strothotte, 1989] developed a prototypical chemistry explanation system which generates pictorial explanations automatically and is capable of leading a dialogue with the user. For example, as an answer to the question "How is N_2 produced?", several pictures showing the steps of the chemical production are presented. In this system, the user can manipulate the labels on the diagrams to obtain more information. However, the flexibility of the dialogues is attained at the expense of the quality of the graphics: Strothotte's system relies on handmade bitmapped images, that is, users are not able to modify the pictures presented.

Besides the research on the generation of information presentation, much work is carried out on algorithmic methods for computing the images to be presented. Improving the performance of image rendering can be done by using visibility precomputations, since by performing work off-line they reduce the effort involved in solving the hidden-surface problem. In particular, many spatial subdivision techniques have been proposed for speeding up ray tracing ([Weghorst et al., 1984], [Glassner, 1989]) as well as for preparing interactive walkthroughs through complex environments, for example ([Teller et al., 1991]). Another method to improve the performance of image rendering is to convey most of the information to the user as early as possible, with image quality constantly improving with time, that is, to render images by adaptive refinement ([Bergman et al., 1986]).

3. Working with a Hyper-Renderer

3.1 The Process of Hyper-Rendering

A software architecture for graphics systems using hyper-rendering is illustrated in Figure 1. The non-shaded boxes with a scene description, a renderer and the resulting picture are as used in conventional graphics systems. Conceptually, hyper-rendering software is built around the renderer. It computes various pieces of information about both the rendering process and the rendered picture, which conventional renderers either throw away or don't bother computing in the first place. This information is stored in an "image description file".

The primary purpose of hyper-rendering is to allow users to do more with the rendered pictures than just look at them; this is facilitated by various kinds of applications which take as input the image description. Depending on the application, the scene description may also be used or modified. An application thus has information about what the user can see in the picture and is able to handle the dialogue with the user about the picture, evaluate the picture under certain criteria or even modify it so as to change the visibility of the objects.

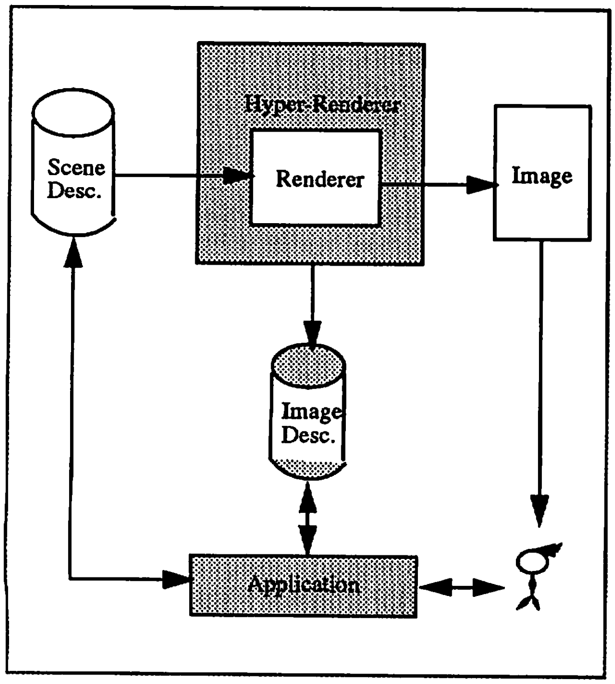


Figure 1: *A new software architecture for graphics systems with integrated hyper-rendering.* The hyper-rendering software is built around the renderer and computes various pieces of information about both the rendering process and the rendered picture, which conventional renderers do not provide for further processing.

3.2 Applications

We consider there to be two broad categories of applications of hyper-rendering. The first category pertains to applications with direct end-user involvement. If a user is to engage in a dialogue about the contents of the graphics with the machine, it is an absolute prerequisite that the application has at its disposal detailed information about what the user can see. This is particularly important in virtual reality or cyberspace applications, when the user will ask for information and explanations about certain objects in the scene. It should be particularly clear here that the scene description itself is not sufficient for the machine for the purposes of leading such a dialogue, since a user's input must be interpreted with respect to what he can see (or in some cases more importantly what he cannot see) rather than only with respect to the model of the scene. The hyper-renderer can also be used in teaching material, when a pupil will ask for information and explanations about certain parts of the graphics.

The second broad application of hyper-rendering is in situations in which a graphic shown to the user (or intended for display) must be evaluated with respect to its appropriateness by the machine. Linking an application program to a renderer without modifying the latter is not possible without storing the information the renderer calculates. With our architecture, it is in particular possible to build knowledge based systems which analyse the modeling of a scene with respect to given constraints, generate improvements, and prepare explanations for the user. For example, [Fischer et al., 1990] discusses the critiquing approach to building knowledge based interactive systems and describes a critiquing system for the 2D design of kitchens. A problem in 3D modeling, in particular when dealing with a virtual environment, is to avoid penetrating rigid objects or having them collide. As standard methods for computing collisions have high computational complexity, [Pentland, 1990] describes more efficient methods for the calculation of dynamics, collision detection, and constraint satisfaction. Another deficiency of scene modeling is when parts of the scene where the observer should focus on lie in shadow. In this case, an improvement generator can suggest better locations and parameters for the light sources.

It is important to note that hyper-rendering is *independent* of the modeling software employed and can be used to improve the modeling process indirectly in contrast to other approaches, where new modelers are developed (for example [Hall, 1991]). This is particularly important, as many deficiencies in scene modeling can only be determined with significant effort by modelers. For example, only a renderer knows where shadows are located and how they influence the perception of the image.

The hyper-renderer can, however, hardly be used to design good (or appropriate) presentations automatically. It can be used in the evaluation of a particular graphic with respect to communicative goals. If the evaluation is negative, other means must be found to correct the situation.

In both these categories of applications, the facilities of a hyper-renderer provide vital input to programs using photorealistic graphics as a communicative tool. The success of such applications depends on the sophistication with which the hyper-renderer works to obtain information on the graphics and its perceptions by the user.

4. A Prototypical Hyper-Renderer

Rather than to integrate a hyper-rendering facility into an existing renderer, we chose to work with a commercially available renderer (in the present case Pixar's RenderMan) and build a hyper-rendering facility as an extra program. While this has the disadvantage that certain code must be duplicated, it allows us in fact to use different algorithms for rendering and hyper-rendering.

Our hyper-renderer is written in about 5k lines of C code on an IRIS 4D35. Input is the scene description file in the format used by the RenderMan. To support maintainability of the scene description file and to facilitate dialogues about the graphics produced by the renderer, we extended its format to include symbolic names of the objects as well as grouping of objects into compound objects.

The hyper-renderer contains typical rendering algorithms which have been enhanced to record information symbolically about the graphics. In particular, an extended z-buffer-algorithm was implemented for hidden-surface removal; however, as opposed to conventional z-buffers, information about hidden surfaces is stored, not thrown away. Our implementation is related to Atherton's implementation of an object-buffer ([Atherton, 1981]). However, while Atherton's three-dimensional display buffer was implemented in the form of a solid object description, we approximated quadric surfaces through polygons. Although the basic algorithmic method used is not new, the *output* produced by the hyper-renderer, namely the image description file, is a significant enhancement. Finally, we used our z-buffer to generate shadows which are caused by opaque objects, as proposed by Atherton as well (see also [Appel, 1967] and [Bouknight et al., 1970]).

It is important to note that the resolution of the hyper-rendering algorithms need not be the same as the resolution of the actual rendered picture; the resolution is determined dynamically by the application program. A coarser hyper-renderer suffices for many applications and means that the results of hyper-rendering can be made available significantly before the rendered picture is in fact available.

By default, our hyper-renderer carries out a fast z-buffer-algorithm in a first pass. By recording which objects are in the line of sight of each pixel, and with information about which surfaces cause, for example, specular reflection or refraction, those parts of the scene which require ray tracing are determined. This way, our ray tracer, which is currently being implemented, will be "selective", that is, restricted to such parts of the picture as are deemed necessary. Furthermore, the visibility information supplied by the z-buffer algorithm can be used for hidden surface removal which makes the ray tracer itself much more efficient (this concept is related to that of an "item-buffer" as described by [Weghorst et al., 1984]). The calling application program can, however, override these "fancy" features and force the use of particular hyper-rendering algorithms.

The output of our hyper-renderer is an image description. It is implemented as a file organized in an object-oriented manner. The names of objects of the scene description file are associated with various pieces of information, including low-level data as to pixels affected by the object and the colors as well as high-level information pertaining to the visibility (and invisibility) of objects, their interior, the intersection of objects, certain prepositional attributes (in front of, behind of, on,...) as well as a description of which objects lie in shadows. Figure 2 summarizes the data produced by our hyper-renderer.

Note that our prototypical hyper-renderer is by no means complete in the sense that is conceivable that new applications will require more or different information. Indeed, the modular implementation of the hyper-renderer supports extensions by a systems programmer. In particular, the complex information alluded to in Chapter 3.2 needs further study before its extraction can be implemented.

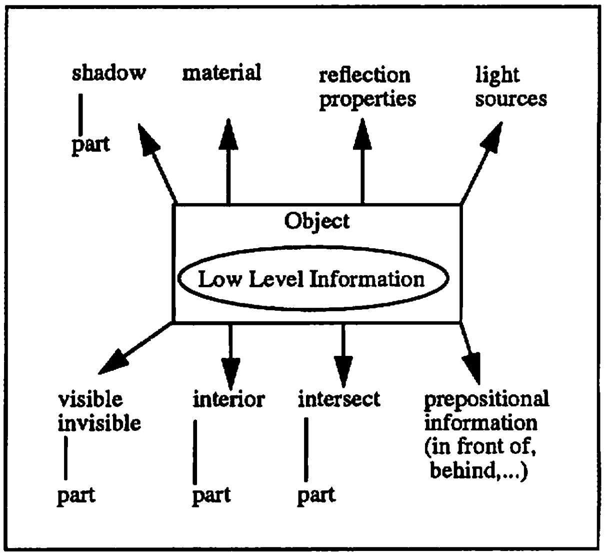


Figure 2: *Information contained in the image description file.* The hyper-renderer produces as output for each object and its parts a semantic net which contains information about visibility and invisibility, shadows, the interior of the object, other objects intersecting it, prepositional information, information about the position of light sources relative to the object, its reflection properties, and its material properties.

5. An Example

5.1 A Prototypical Application

In order to demonstrate the capabilities of hyper-rendering, we developed a simple dialogue system for navigating in a scene. The dialogue is conducted in a restricted natural language which allows a user to specify the kind of graphics to be displayed (wire-frame or full-surface-rendering). The most significant feature of the application is that it allows the user to formulate a constraint on the view, upon which the application computes a change in the scene description file and initiates re-rendering and re-hyper-rendering of the graphics.

5.2 The Dialogue and a Sample Session

We will explore the interior of an IKEA cupboard. A description of the cupboard and its contents was designed with a modeler and stored in a RenderMan "rib"-file. The end-user now uses the application to draw the picture by typing an appropriate command:

Application:	Please enter command.
User:	Draw cupboard.

The RenderMan full-surface renderer is invoked by the application and produces Figure 3.

The user now wishes to see more of the object in the picture and decides to switch to a wire-frame image.

Application:	Please enter command.
User:	Draw wires.

The RenderMan is invoked again and produces Figure 4.
The user realizes that the wire-frame image does not provide enough information to find out what is in the cupboard. Hence he asks for information on what he cannot see:

Application:	Please enter command.
User:	What can I not see?
Application:	Ball, box, safe.

In this latter response, the application has made use of the hyper-renderer tool. The image description contains information about the visibility of objects and from this it is easy to compute which objects are *not* visible.

The user now wishes to look at a particular object, the safe. By looking at the wire-frame image, he recognizes that there are *two* candidate objects which can be a safe (one is on the bottom and another is below the top on the left side). Hence, he enters the following command:

Application:	Please enter command.
User:	Show safe through glass.

The application responds with Figure 5.

The application used the information in the image description file to determine that it was the cupboard itself which blocked the user's view of the safe. It then changed the material of which the upper left part of the cupboard is made to "glass" in the scene description and drew the resulting picture.

5.3 Résumé

The dialogue illustrated above is a simple example of the use of hyper-rendering. While it is in principle possible to extract such information as "What can I not see?" directly from the scene description with a significant amount of work, we believe our hyper-rendering to be the first general purpose *tool* for interactive graphics which allows an application to determine this kind of information in a simple manner. Furthermore, implementing a command such as "Show safe through glass", specifying a constraint on the visibility of a scene is easily accomplished when a hyper-renderer is available but would be tedious to program without such a tool.

6. Concluding Remarks

In this paper we introduced the concept of hyper-rendering which makes information about the rendered image available to an application. We argued that such information is useful for providing information about the graphics to end-users as well as to evaluate the graphics displayed. We demonstrated the important facilities of the hyper-renderer with a sample dialogue application.

The concepts we introduced open up a range of new problems for further study. One area is to use hyper-rendering to improve the rendering process itself. As the performance of the rendering process can be improved by refinement, that is, by subdividing it into phases where the results of each phase are carried over to the next phase ([Bergman et al., 1986]), hyper-rendering could be used to supply additional formalized information which in turn speeds up the performance of a succeeding phase.

In addition to this, hyper-rendering provides a tool for developing more effective multi-media dialogue systems. Since more information about the graphics is available as a result of hyper-rendering, speech or text interaction concerning or integrated with the graphics has a better chance of being appropriate. In particular, hyper-rendering can offer additional interaction facilities and information for users who are directly placed in a virtual environment or cyberspace. For example, users could ask questions about their location and the objects they are looking at. Therefore, hyper-rendering could prevent users from being "lost in cyberspace". Furthermore, if users have difficulties in interpreting the graphics output by the machine, it is in turn possible to *train* them to see and to guide them where to look. For example, during an interactive presentation it is often difficult to look at the "right" objects displayed, as humans are, in general, not trained in this capability. Using our hyper-renderer, it is possible to develop interactive multi-media systems which allow users to lead dialogues about the graphics with the goal of discovering what is important.

7. Acknowledgments

We would like to thank Walther Beck for various helpful discussions on aspects addressed in this paper and Petter Ranefall for implementing the prototypical hyper-renderer and application described. We would also like to thank an anonymous referee for some very detailed comments on the initial draft of the paper.

8. References

1. A. Appel, The Notion of Quantitative Invisibility and the Machine Rendering of Solids, in *Proc. ACM, Vol. 14*, pp. 387-393, 1968.

2. P.R. Atherton, A Method of Interactive Visualization of CAD Surface Models on a Color Video Display, in *Computer Graphics, Vol. 15, No. 3*, pp. 279-287, August, 1981.

3. L. Bergman, H. Fuchs, E. Grant, S. Spach, Image Rendering by Adaptive Refinement, in *Computer Graphics, Vol. 20, No. 4 (Proc. SIGGRAPH'86)*, pp. 29-34, August, 1986.

4. W.J. Bouknight, K.C. Kelley, An Algorithm for Producing Half-Tone Computer Graphics Presentations with Shadows and Movable Light Sources, in *SJCC, AFIPS, Vol. 36*, pp. 1-10, 1970.

5. S. Feiner, APEX: An Experiment in the Automated Creation of Pictorial Explanations, in *IEEE Computer Graphics and Applications, Vol. 5, No. 11*, pp. 29-38, November, 1985.

6. G. Fischer, A.C. Lemke, Th. Mastaglio, Using Critics to Empower Users, in *Proceedings of the CHI '90 Conference on Human Factor in Computing Systems*, pp. 337-347, April, 1990.

7. J. Foley, A. van Dam, S. Feiner, and J. Hughes, *Computer Graphics: Principles and Practice, 2nd Edition*. Addison-Wesley, Reading, MA, 1990.

8. A.S. Glassner, *An Introduction to Ray Tracing*. Academic Press, San Diego, CA, 1989.

9. R. Hall, M. Bussan, P. Georgiades, D.P. Greenberg, A Test-bed for Architectural Modeling, in *Proc. EUROGRAPHICS'91*, Vienna, pp. 47-58, September, 1991.

10. J. Mackinlay, Automating the Design of Graphical Presentations of Relational Information, in *ACM Transactions on Graphics, Vol. 5, No. 2*, pp. 110-141, April, 1986.

11. A.P. Pentland, Computational Complexity versus Simulated Environments, in *Computer Graphics, Vol. 24, No. 2*, pp. 185-192, March, 1990.

12. D. D. Seligmann, S. Feiner, Automated Generation of Intent-Based 3D Illustrations, in *Computer Graphics, Vol. 25, No. 4 (Proc. SIGGRAPH'91)*, pp. 123-132, July, 1991.

13. Th. Strothotte, Pictures in Advice-Giving Dialog Systems: From Knowledge Representation to the User Interface, in *Proc. Graphics Interface'89*, London, Ontario, pp. 94-99, June, 1989.

14. S. J. Teller, C. H. Sequin, Visibility Preprocessing for Interactive Walkthroughs, in *Computer Graphics, Vol. 25, No. 4 (Proc. SIGGRAPH'91)*, pp. 61-69, July, 1991.

15. W. Wahlster, E. André, Som Bandhyopadhyay, W. Graf, T. Rist, WIP: The Coordinated Generation of Multimodal Presentations from a Common Representation, in *Computational Theories of Communication and their Applications*, Oliviero Stock, John Slack, Andrew Ortony (eds.). Springer-Verlag, Berlin, 1991.

16. H. Weghorst, G. Hooper, D.P. Greenberg, Improved Computational Methods for Ray Tracing, in *ACM Transactions on Graphics, Vol. 3, No. 1*, pp. 52-69, January, 1984.

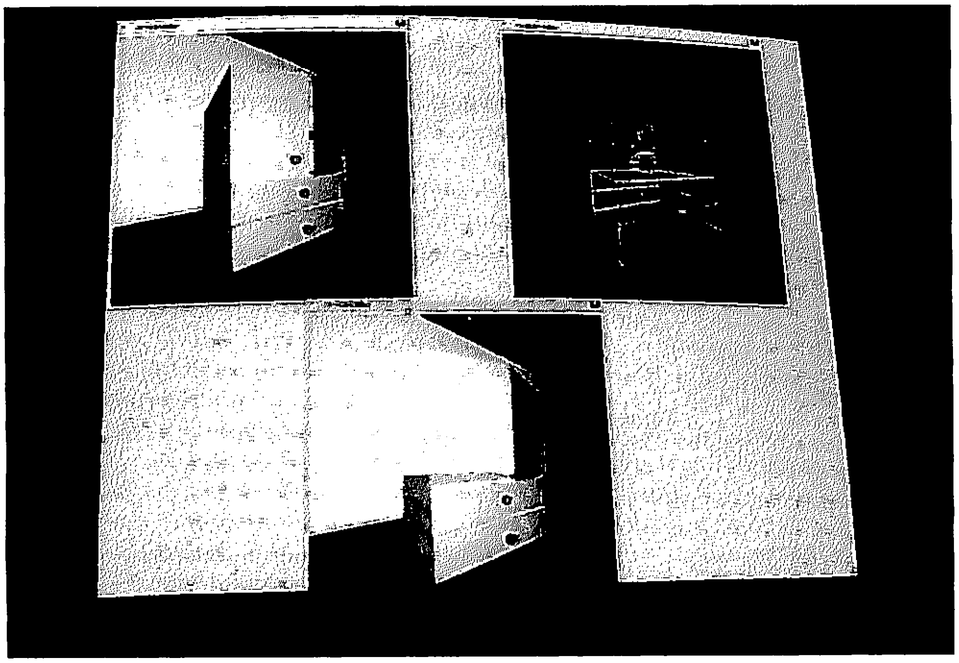


Figure 3 (top left): *The rendered image of the IKEA cupboard.*
The user is unable to see many objects, such as those lying inside or behind the cupboard.
Figure 4 (top right): *The rendered wire-frame image of the cupboard.*
All objects are visible, though largely indiscernible.
Figure 5 (bottom): *The modified image showing the inside of the cupboard.*
The user previously recognized that there were two candidates for the safe in the wire-frame image. He thus entered the command "Show safe through glass". Based on the information placed at the disposal of the application by the hyper-renderer, the application converted the material of which the upper left part of the cupboard is made to glass, thereby offering the user a view of the objects of interest to him.

Program Auralization: Sound Enhancements to the Programming Environment

Christopher J. DiGiano
digi@dgp.toronto.edu

Ronald M. Baecker
rmb@dgp.toronto.edu

Dynamic Graphics Project
Department of Computer Science
University of Toronto
Toronto, Ontario, Canada M5S 1A4

Abstract

Sound has the potential to improve our understanding of a program's function, structure, and behavior. In this paper we identify classes of program information suitable for mapping to sound and suggest how to add auralization capabilities to programming environments. We describe LogoMedia, a sound-enhanced programming system which illustrates these concepts.

Keywords: Program Auralization, Non-Speech Audio, Software Visualization, Programming Environments

Program auralization

Gerald Weinberg, author of The Psychology of Computer Programming, describes programming as "at best a communication between two alien species" (Weinberg, 1971). Indeed, despite efforts in the field of software visualization, programs are in desperate need of better means of presentation and clarification. This paper explores the potential of non-speech audio to increase the communications bandwidth between the two "species."

Non-speech audio is quickly becoming an integral part of the computer's ability to record and present data, as is evidenced by recent introductions of computer systems with built-in high quality sound input and output from graphics workstation manufacturers such as Sun Microsystems (Yager, 1991) and Silicon Graphics (Smith, 1991). Interface designers have only just begun to tap into the capabilities of this new hardware. Although sound has seen limited use in some software visualization systems (Baecker, 1981; Brown, 1988), it is only recently that computer audio has been seriously applied to the programming domain to help elucidate the behavior of running programs (Jackson and Francioni, 1992; Sonnenwald, et al., 1990). *Program auralization* refers to the use of non-speech audio for supporting the understanding and effective use of computer programs.

The first part of this paper discusses some properties of sound that are useful to the human-computer interface. We then propose a program auralization taxonomy describing how sound may be associated with execution behavior and with the structure of the code. For each of the three categories in the taxonomy we discuss the relevant characteristics of sound and useful types of program auralization tools. We conclude by describing our implementation of a sound-enhanced Logo programming environment called LogoMedia.

Non-speech audio at the interface

Sound has unique properties which can be exploited in the computer-human interface. For certain types of data sound is the most intuitive means of understanding the information. Experiments mapping time-varying economic data to sound have found that humans can more effectively identify correlations using sound than with graphics (Mezrich, Frysinger, and Slivjanovski, 1984). Logarithmic data, normally difficult to perceive graphically, has also been shown to benefit from audio portrayals since pitch and loudness are logarithmic functions of frequency and intensity (Buxton, Gaver, and Bly, in preparation, ch. 3). Gaver has demonstrated the ability of "everyday" sounds to alert users instantly yet unobtrusively about certain events (Gaver, 1989; Gaver, 1990; Gaver, 1991a; Gaver, 1991b).

Sound can be used to relieve the burden of the visual interface. With the increasing popularity of graphical interfaces, more and more applications are using images to portray information. The resulting collage of windows and colors has the potential of overloading the human visual system. The computer industry is also moving toward smaller, more portable computers with displays limited by current technology to fewer colors, less pixels, and slower update rates. The effective use of sound offers an attractive design solution to problems introduced by both trends, providing a new output modality to complement the graphical interface.

The multi-dimensionality of sound makes it useful for presenting complex data which is otherwise difficult to represent. Sound can be systematically varied across many dimensions including loudness, pitch, vibrato, rate of modulation, timbre, and tempo (Buxton, Gaver, and Bly, in preparation). Some works (Yeung, 1980) suggest human can differentiate sounds of up to 20 dimensions. Another useful property of sound is that humans do not necessarily have to be facing the source in order to hear it. This means that the focus of attention can be applied to the computer screen, a printout, or even a cup of coffee while continuing to process auditory information.

A program auralization taxonomy

Auralizations appropriate to the programming domain can be divided into three major categories according to the characteristics of the information streams mapped to sound sequences. Each sound generated as part of an auralization corresponds to some item in a time varying stream of data. We characterize these streams by their *sound generators*—the activity which determines which datum comes next for the purpose of auditory mapping. Probably the most obvious information stream is the sequence of program states during execution. In this case the sound generator is the running program itself. As shown in Table 1 we have identified two other unique sources of program sounds. Each activity corresponds with a program development phase and covers a particular type of program information.

Phase	Sound Generator	Program Information
execution	executing	variables, internal state, control flow, backtracking
review	scrolling	modules, goals/plans, keywords
preparation	parsing	syntactic structure

Table 1: *Three categories of program information which are candidates for auralization*

Execution

The execution of a program causes variables and machine state to change over time. By mapping this data stream to sound the programmer can listen to his or her code run which has the potential to reveal useful information about the behavior of the program.

Review

A programmer examines his or her code by scrolling through the text in a window. The sequence of program sections encountered can be mapped to sound, helping provide contextual information while using the visual information from the screen for detailed assessment.

Preparation

While compiling or entering a program, the computer evaluates language tokens and identifiers in a specific sequence not necessarily corresponding to the stream of characters making up the code. This sequence can be translated into sound, allowing the programmer to monitor the progress of such activity.

An Example

Consider the following example programming session in which we auralize the process of preparing, executing, and reviewing a procedure to compute the factorial function. Figure 1 shows one way of performing this calculation using Logo (Harvey, 1986). A single call to factorial generates a series of nested recursive calls which multiplies a series of numbers decreasing by one from the original value to one.

```
1  to factorial :num
2    if :num = 0
3      [output 1]
4      [output (:num * factorial (:num-1))]
5  end
```

Figure 1: *Logo code for computing factorial*

While typing line 4 we must be careful to match all of the nested delimiters. Using a standard editor for entering the code, we could easily skip one of the right parentheses. A sound-enhanced programming environment in which sounds are generated during program entering can inform us immediately of such an error. Typing the left bracket might initiate an unobtrusive continuous background sound such as the soft ticking of a clock as depicted in Figure 2. The following left parenthesis might start the background sound of a bubbling fish tank which is layered on top of the ticking noise. A fan noise might result from typing yet another left parenthesis. Each background sound continues playing until its respective matching delimiter is typed, so that by the end of the line none of the three channels should be audible. Now if we skipped one of the right parentheses we would reach the end of the line without the bubbling stopping—an obvious indication of a problem.

Suppose we mistakenly implemented factorial as shown in Figure 3 without the base case, resulting in a program which runs forever or at least until stack memory is exhausted. When we call our flawed factorial procedure from the Logo interpreter with the argument of 20, the computer would pause for a while and finally produce an error message. Suppose we suspected the problem was actu-

```
[                                        ticking
[output (                                ticking and bubbling
[output (:num * factorial (              ticking, bubbling and fan
[output (:num * factorial (:num-1)       ticking and bubbling
[output (:num * factorial (:num-1))      ticking
[output (:num * factorial (:num-1))]     silence
```

Figure 2: *Delimiter matching sounds while entering line 4 of the procedure in Figure 1*

ally with the value of the parameter `:num`. Was it actually decrementing for each successive call, or was the computer using the outermost runtime stack frame to determine the value of `:num`? Using a sound-enhanced programming environment we can monitor the execution behavior of this and other parameters. We might associate a continuous sound which varies in pitch with the value of `:num` as the program executes—the lower the number, the lower the pitch. Now when we execute the program, we would hear a note which starts high but quickly drops in pitch. It would be apparent that `:num` is indeed being decremented and our error must be elsewhere.

```
to factorial :num
    output (:num * factorial (:num-1))
end
```

Figure 3: *Factorial procedure with missing base case*

If the factorial procedure were part of a larger program it might help us to understand the code if we knew how often factorial gets called and by which components. A sound-enhanced programming environment can be asked to generate a sound each time the identifier `factorial` scrolls through the editor window. The sound warns us in advance to pay closer visual attention to the code coming into view.

Related audio applications

In recent years the use of non-speech audio at the interface has become a serious study for interface designers. A seminal work by Gaver (1989) was the Sonic Finder which used everyday sounds to indicate common operations in the Macintosh direct manipulation interface. Dragging the mouse generated a scratching sound. The act of deleting files by dropping their desktop icons into the trash emitted a crash. The mouse cursor at times also acted like a mallet, creating sounds as the user clicked on various icons on the screen. Large file icons had a low hollow sound, while smaller files had a higher pitch.

Following the Sonic Finder were ARKola (Gaver, 1991a) and EAR (Gaver, 1991b) which used sophisticated event-driven sounds. ARKola simulated the sounds of a collection of bottling plant machines running simultaneously in various stages of disrepair. EAR sounds included paper falling, the shuffling of people gathering in a room, and the pouring of a pint of beer to remind users of various events and conditions taking place or about to take place at Xerox EuroPARC.

Other studies have focused on using sound to represent complex computer data, referred to as *data auralization.* Exvis (Smith, Bergeron, and Grinstein, 1990) used both graphics and sound to portray various types of data including magnetic resonance scans which varied across many dimensions. As the user moved the cursor over the graphical representation, data points emitted a characteristic sound with frequency and intensity related to particular dimensions of the sample. Bly (1982) tested human analysts on six-dimensional data which she mapped to six sound characteristics: pitch, volume, note duration, fundamental waveshape, attack envelope, and overtone waveshape. She found that the information content of the data could indeed be enriched through audio.

As an aural alternative to graphical icons Blattner, Sumikawa, and Greenberg (1989) proposed "earcons" for denoting a variety of computer events. Earcons consisted of single notes, short melodies or combinations of other earcons. The authors suggested that properly designed earcons could be learned quickly and associated with arbitrary objects and computer operations.

Recent works have demonstrated the potential of audio in the programming domain. Jackson and Francioni (1992) used audio to improve the programmer's awareness of the behavior of parallel programs by generating sounds based on trace data recorded during execution. Program events being monitored caused a unique note or melody to be played using a particular timbre mapped to each of 16 processors. Music theory was used to identify the most appropriate melodies for each event. Sonnenwald, et al. (1990) developed a set of primitive function calls for incorporating sound in program code for the purpose of elucidation. Although the authors primarily discussed the capabilities of their system for portraying parallel programs, its general audio functions could be applied to a variety of programs both concurrent and non-concurrent. While animating algorithms Brown and Hershberger (1991) generated sounds corresponding in pitch to elements being inserted into a hash table, items being sorted, and to the number of active threads. The sound of a car crash we used to indicate hash collision. Brown and Hershberger identify four main uses of sound in algorithm animation: "reinforcing visual views, conveying patterns, replacing visual views, and signaling exceptional conditions."

Sound in the programming domain

Most of the above examples deal with sound in specific applications. In contrast this paper discusses the uses of non-speech audio in a much more generic context—the programming environment. Auralization in the programming environment is a challenging interface design problem since the sounds must be adaptable to a variety of conditions. A sound-enhanced programming environment must represent both data and events aurally. A further complication is that the exact types of data and events are unknown, since the programming environment is meant to be used to develop arbitrary applications using a variety of data and control structures. Finally, depending on the stage in the program development process, the programmer may use sounds in different ways. When debugging, for instance, the programmer may want to introduce generic or symbolic sounds quickly to determine the behavior of his or her program. When creating auralizations for presentation purposes, he or she may want more specific or iconic sounds to portray more appropriately the program execution.

Figure 4 compares non-speech audio applications based on the type of information they represent using sound as well as their *articulatory directness.* Articulatory directness as defined by Hutchins, Hollan, and Norman (1986) is the degree to which form follows function in an interface. Gaver (1989) used this measure to describe the intuitiveness of the perceptual mappings used to link the "model world" of the computer and the audio display, with *symbolic* being the least intuitive and *iconic* the most. A symbolic mapping is an arbitrary association made between sound and computer information, whereas an iconic mapping is based on well-known physical properties. Exvis and Bly's work which focus on data auralization belongs near the symbolic end of the articulatory directness scale because they relate properties of sound such as pitch and loudness to data which has no inherent connection with sound. ARKola and EAR on the other hand use iconic mappings because of the obvious associations between events and their sounds.

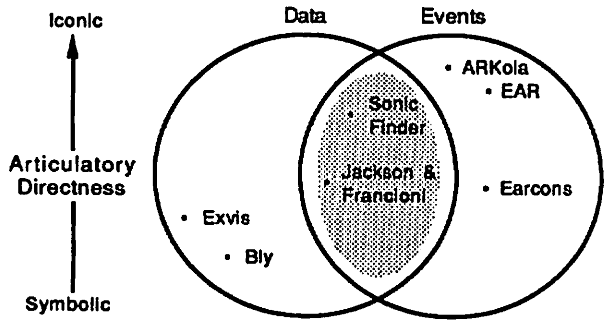


Figure 4: *Comparison of non-speech audio applications*

The intersection of the two circles in the figure contains those applications using non-speech audio to represent both data and events. The Sonic Finder belongs in this category because of the way impact sounds suggest the type and size of files. Another application for this category is program auralization systems which portray both data and control flow using sounds. A sound-enhanced programming environment, represented by the shaded area of the figure, must use sound to portray both data and events. An area is more appropriate than a point because the programming environment must be capable of using sound to portray a variety of different types of data and events at different levels of articulatory directness.

Execution sounds

Comprehending the course of execution of a program and how its data changes is essential to understanding why a program does or does not work. Auralization expands the possible approaches to elucidating program behavior.

Protocol studies of novice, intermediate and expert programmers debugging code suggest they make use of a variety of techniques for observing the changing values of variables and monitoring sub-program calls (Nanja and Cook, 1987). As shown in Table 2 we can divide execution information into values and events. Value information refers to the contents of data structures as they change during execution, often known as data flow. Event information is the stream of operations on these data structures as well as the flow of control from one line of code to the next and from one sub-program to the next.

Information Type	Values	Events
common	queue size, tree depth	loop, branch, push, pop, search, sort
arbitrary	a, b, c	sub-program calls, sub-program returns
internal	call stack size, memory space	register usage, backtracking

Table 2: Candidates for auralization during execution

For both values and events we group the execution information into three categories: common, arbitrary, and internal. *Common* information refers to typical data and control structures which we can anticipate will be the subject of interest to the programmer. Sound enhancements to the programming environment should include default methods for monitoring the status and usage of these structures. *Arbitrary* execution information refers to the types of data and control flow which cannot be predicted. For this type of information a sound-enhanced programming environment should provide tools for establishing custom auralizations. *Internal* information is the changing machine state over the course of execution which is largely programming language and machine dependent. Default auralization for this type of data can increase the programmer's awareness of activities such as resource usage occurring behind the scenes.

Monitoring execution values

Traditional techniques for monitoring values during execution generate text to indicate the current program state. An obvious method of displaying text relating to program variables is to insert output statements directly into the source code. Debuggers allow the values of variables to be printed without having to modify the source code. A popular debugger for the Unix environment, dbx provides a set of commands for maintaining a list of expressions containing program variables to be printed during execution.

Software visualization systems use pictures to present often more complex program data in ways which are simply not possible using text. Visualizations are useful for making abstract program information easier to understand. The University of Washington Program Illustrator, for instance, recognizes the abstract data structure for digraphs within Pascal programs and represents them on the screen as a collection of circles connected by arrows (Henry, Whaley, and Forstall, 1990). Pictures of programs are well-suited for displaying large quantities of computer information simultaneously. In the film *Sorting Out Sorting* Baecker (1981) uses graphs to show the progress of several sorting algorithms on thousands of data items.

Data auralization provides a useful extension to textual and graphical techniques for monitoring execution. Although not suitable for conveying exact values, auralization can indicate trends and increase the number of dimensions capable of being presented simultaneously. Auralizations of program data generate sounds varying across dimensions of pitch, volume, reverberation, panning, envelope, and tempo in response to the changing values of program variables. An important advantage over visual feedback is that auralization frees the programmer to inspect the actual code while it is executing.

Sound can be useful in conveying simple auxiliary information relating to common data structures such as the size of a queue or the depth of a stack. Ideally, these auralizations of high level data structure values could be built into the language or class library. Thus, the user is free from having to specify the particular element of the data structure to map to sound.

For listening to arbitrary data a sound enhanced programming environment must support the ability to map changes in variable values to a variety of audio dimensions. The mapping must be flexible so that the user can experiment with various auralizations in order to find the one most appropriate for his or her application. Users should be able to select from a variety of synthesized and prerecorded sounds, or make their own sounds. To support this type of opportunistic auralization in LogoMedia we have developed an audio expressions tool for making data-sound associations.

Internal values in an executing program refer to machine states not normally known to the programmer. The size of the call stack and the amount of free memory are two examples of internal values. The default sounds available for internal values should easily fade into the auditory background of the listener. Patterson's study of alarms (1989) provides useful guidelines for designing non-obtrusive sounds such as the use of slow onsets and rhythmic patterns.

Monitoring execution events

Traditional techniques for monitoring execution events produce a sequence of text lines indicating control flow. The simplest method requires no additional programming environment tools: inserting lines of code before or after suspect statements which cause a message to appear on the screen. More sophisticated program tracing approaches are found in debuggers such as dbx which can automatically generate generic tracing information while simulating the execution of a program.

Software visualization systems use pictures or diagrams to indicate the execution path. Some visualization systems such the Transparent Prolog Machine (TPM) (Eisenstadt and Brayshaw, 1988) can graphically trace programs automatically. Other software visualization systems offer more customizable and therefore potentially more salient trace feedback than traditional debuggers. LogoMotion (Baecker and Buchanan, 1990) is a programming environment in which procedure calls can trigger tailored visualization code.

Computer audio can enhance textual and graphical execution event information by generating sounds in conjunction with the execution of a line or a collection of lines and by using multiple voices to indicate layers in a calling chain. As with program data, control flow auralization allows the programmer to focus his or her visual attention on the actual code being run. Tracing programs using audio can also uncover repeated patterns of program behavior. Jackson and Jackson (1992) noted that users of their system for auralizing parallel program events recognized "melodies" characterizing various communication patterns between processors. They also pointed out that missing or delayed parts of the pattern were noticeable.

A sound enhanced programming environment should allow users to associate sounds with common, arbitrary, and internal events. Common events include the execution of control structures such as loops and branches as well as operations on common data structures such as stacks and linked lists. For LogoMedia we have developed sounds of plates stacking, unstacking, and breaking for the typical push, pop, and overflow events. Additionally, there are sounds for indicating list operations using the metaphor of a three-ringed binder—popping open for an insertion, tearing for a deletion, and page flipping sounds for a search.

Arbitrary events of interest to the programmer might be calls to sub-programs, returns from sub-programs, or the execution of particular lines. Sound enhancements to the programming environment should include the ability to generate sound in response to the program counter reaching arbitrary lines of code. In LogoMedia entering or exiting a procedure can trigger auxiliary Logo code for turning sounds on or off or changing their quality. We are currently developing a tool for specifying program *audiopoints* instead of breakpoints.

The ability to monitor language or machine dependent internal events using sound is certainly worthy of further study. Low level internal information such as operations on registers and high level information such as Prolog backtracking may prove to be good candidates for auralization, since the programmer is more likely interested in the general pattern of activity, rather than exact events.

Review sounds

The review phase when the programmer interactively explores the source code is another opportunity for integrating sound into the programming environment.

Studies in program comprehension (Gellenbeck and Cook, 1991; Kesler and Uram, 1984) have reported the utility of information supplementary to the raw code in providing clues to help programmers understand a program and predict its behavior. Examples of this kind of information include indentation, comments, typographic signaling, mnemonic names, module organization, goals and plans, profiling data. Sound offers a new modality for communicating this ancillary information. Unique to this phase is the method by which sequences of sounds are derived from the series of points in the code on which the programmer focuses his or her attention.

Visualizations of software can enhance the programmer's understanding of programs by providing a variety of views which either compress or elide uninteresting information. Baecker and Marcus illustrated a collection of information-rich typographic overviews in SEE (Baecker and Marcus, 1990). Small's static typographic visualization system, Viper, revealed the advantages of interactive systems for culling, clarifying, and amplifying parts of the program text interesting to the user (Small, 1989). Viper allowed the programmer to specify code filters used for remove certain lines of code and highlight statements in others.

Listening to source files

Computer audio offers an attractive alternative to textual enhancements to the code for conveying information about the structure or intent of a program. Sound-enhanced programming environments should allow portions of code at the focus of the user's attention to be mapped to sound. Perusing the code causes the system to generate a sequence of sounds, providing an auditory context for more detailed visual examination.

Auralizing the source code requires that interesting portions of the text somehow be identified for mapping to sound. This can be done automatically by parsing the code and flagging syntactic structures such as blocks, and procedures as is done with Viper. Profiling and verification systems such as Unix `prof` or `lint` can automatically associate other kinds of useful data with sections of code. A semi-automated alternative is using a structured editor which can identify syntactic structures and problem solving strategies. Finally, the programmer can manually select the sections of code for auralization. Although this last option may seem tedious, the programmer may be able to take advantage of auralizations already in place for the purpose of monitoring events during execution. If the programmer had been thorough in marking a variety of interesting program events with audiopoints, the events themselves can serve as "audio landmarks" (Jenkins, 1985) into the code—a form of audio documentation.

In LogoMedia we are developing the ability to generate audio feedback while scrolling through a Logo document. As interesting items enter and leave the window view the sound will change accordingly. Program constructs identified during interpretation such as procedure definitions, Logo lists, and lines with audiopoints will each have characteristic sounds. Programmers can perform an auditory search of their code by asking the system to generate sounds if the items in focus meet certain criteria such as if they are an identifier of a certain name or a particular Logo command.

Rapid scrolling through a program may generate emergent sounds patterns indicative of particular types of code or a certain style of programming. Speeth (1961) found that subjects were able to discriminate between earthquakes and explosions using compressed auralizations of large quantities of seismic data which were not visually distinct. Because of sound's ability to reveal patterns in massive quantities of data, new insights may result from the rapid audio review of complex program.

Preparation sounds

The process of preparing code for execution can be mysterious to the programmer. In a sound-enhanced programming environment the order in which programming language constructs are parsed can be a source of sound sequences and increase the programmer's awareness of this ongoing activity. While entering a program syntax-directed audio feedback can help identify syntactic errors. During compilation sounds can be used to monitor progress.

Syntax-directed editors attempt to reduce obvious syntactic errors through typographic feedback during code entry. In some systems keywords recognized by the editor change font after the programmer types a separator such as the space bar. Lines following an IF clause are automatically indented until the block is terminated with a semi-colon. An obvious problem with syntax-directed editors is that they force users to adopt a particular typographic style. The persistence of typographic syntax-directed feedback may also be an annoyance to programmers. The style of the program text remains changed even after the words have been typed and the programmer is sure it has been entered properly.

Listening to program entering

A sound-enhanced programming environment offers the programmer an alternative output modality for conveying similar syntactic feedback during program entering. An approach we are investigating for LogoMedia is to provide subtle background sounds to indicate whether the computer recognizes program constructs which are typed and to differentiate aurally between classes of syntactic structures. Word classes might include operators, built-in procedures, control constructs, previously declared procedure identifiers, previously declared variable identifiers, new procedure identifiers, and new variable identifiers. Discrete sounds are triggered by completing a typed word with a delimiting character such as the space bar or comma. After a fixed period of time the sound terminates. Following Patterson's approach to reducing obtrusiveness (Patterson, 1989), each sound conforms to an intensity envelope in which the sound begins quietly, is sustained for brief moment, then fades to silence. We are also devising continuous sounds to demark sections of code and help identify problems such as wrong number of delimiters or wrong number of parameters. When the programmer starts a Logo block with a left bracket, for instance, a sound is generated until the matching bracket is typed. Nested sections of code cause collections of continuous sounds to be layered.

Listening to compilation

A programmer typically glances periodically at textual output from a compiler during the translation process. A sound enhanced programming environment should support the ability to generate sounds in response to a compiling program to prevent the user from having to shift visual focus from other activities. In some programming environments such as the Macintosh Programmers Workshop sounds are used to differentiate successful from unsuccessful compiles. However, more sophisticated uses of sound can undoubtedly convey even more than binary status information. In the "early days" of computing intrepid hardware hackers on the TX-2 attached a speaker to the index registers. The resulting sound with some training was used by programmers to determine the state of running processes such as compiles which had limited visual feedback. Richer sounds could be based on the same types of audio feedback used during the interactive parsing of programs as mentioned in the previous section. An interesting question is whether meaning could be extracted from the emergent sounds generated by this high speed parsing of program structure. Since Logo is an interpreted language, LogoMedia has no facilities for compilation-based sounds.

LogoMedia

We have implemented some of the auralization techniques outlined above relating to the execution and review of programs. Sound capabilities were added to a programming environment for software visualization developed here at the University of Toronto called LogoMotion (Baecker and Buchanan, 1990). The goal of the new version dubbed LogoMedia is to provide the programmer with simple, non-invasive[1] techniques for using sound to aid in the development and presentation of arbitrary programs. We are not trying to enforce a particular use of sound for auralizing programs, but rather provide a variety of tools for the programmer to use sound easily while developing.

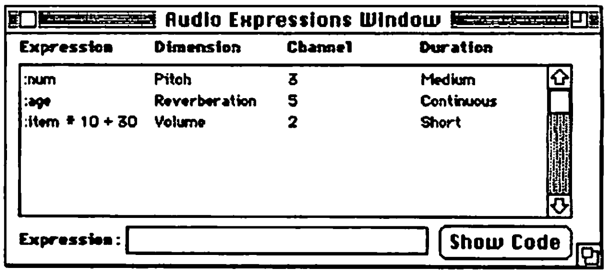
LogoMedia does not actually synthesize its own sounds but rather sends Musical Instrument Digital Interface (MIDI) (IMA, 1983) messages to the Macintosh MIDI Manager which then relays commands to sound generating devices attached to the computer. Using MIDI gives us the flexibility which we believe is essential for auralizing arbitrary programs. The user can generate simple musical sounds through a primitive synthesizer responding to MIDI commands or sophisticated sampled sounds by connecting a different device. Separating the sound generation from the programming environment using MIDI has reduced the possibility that the auralization itself will cause problems in a program being debugged.

[1]Invasive is the term used by Price, Small, and Baecker (1991) to categorize software visualization systems which require modification to the source code. It seems an appropriate descriptor for program auralization systems as well.

LogoMedia has three new commands for generating sound: `startnote`, `playnote`, and `stopallnotes`. `startnote` begins playing a MIDI sound of a given channel, note number, and volume. To turn off a note, the volume of zero is used. The `playnote` command has the additional parameter of duration specified in 60ths of a second. `stopallnotes` turns off sound in all 16 MIDI channels.

Specifying the monitoring of execution values

In order to facilitate the rapid association of sound with execution behavior we have devised a tool for mapping expressions to sound qualities. The audio expression tool tells LogoMedia what sounds to generate during execution based on changes in the program information. Each line in the table describes how an expression containing one or more program variables is related to particular sound qualities of a specified voice. The tool as depicted in Figure 5 was designed to encourage users to use sound in ways which have been successful in previous auralization applications. Specifically, the audio expressions tool makes it easy to differentiate by timbre a collection of runtime values being monitored and discourages users from mapping this data to more than one audio dimension.



Audio Expressions Window

Expression	Dimension	Channel	Duration
:num	Pitch	3	Medium
:age	Reverberation	5	Continuous
:item * 10 + 30	Volume	2	Short

Figure 5: *LogoMedia's expression-sound table for observing changes in program variables*

To prepare a runtime auralization using the audio expression tool the programmer enters a Logo expression containing variables used in his or her code. We chose to map sounds to expressions instead of simply variables to allow the user to scale and offset their values to fit the range expected by the MIDI devices. Pitch, for instance, is specified by MIDI with a value between 0 and 127, yet many instruments, samplers in particular, cannot play notes which cover this entire range. After entering the expression programmer then selects an audio dimension to which the value of the expression will be mapped. Currently, the only available dimensions are pitch and volume, although we plan on adding the ability to control reverberation, stereo panning, and envelope. The next step for the programmer is to select on of 16 MIDI channels which determines the timbre of the resulting auralization for the expression. Lastly, the programmer specifies the duration of the sound as either "continuous" or one of three relative time periods—short, medium, and long. The continuous selection implies after the initialization of any variable in the expression the MIDI channel will play until the program terminates. The relative periods have varying duration depending on the rate of execution selected by the user.

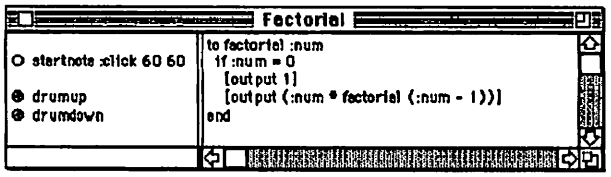
Because of the ease of creating meaningless cacophony, the audio expression tool was deliberately designed to constrain the user to auralization techniques that show the most promise. The layout of the window, for instance, encourages users to differentiate their expressions by timbre and discourages mapping its value to different sound channels. Additionally, the design of the tool makes it difficult to monitor the same expression using more then one audio dimension. A more flexible interface which allowed expressions to be entered for each sound quality including timbre was rejected, since it provided no guidance as to the more effective uses of sound. We have also rejected the ability to enter absolute duration times, since this reduces the adaptability of the auralization to various rates. We believe that the capability to play back programs at different speeds we believe is key to deriving meaning from auralizations and this is planned to be a focus of our user study.

The constraints imposed by the audio expressions tool reflect simply techniques which have been effective in the past, but we cannot expect the tool to facilitate all useful "sonifications." However, the audio expressions tool can be used to generate a first approximation of the appropriate auralization which can later be refined by the programmer using the Logo language. By pressing the "show code" button the user can see the underlying Logo code responsible for auralizing a particular Logo expression. This code can then be copied into a LogoMedia document and edited.

Informal observations from the use of the audio expressions tool reveal weaknesses which must be addressed in the next iteration of the interface before user testing. Scaling and offsetting a variable to fit MIDI protocol has proved cumbersome. A more fundamental problem is predicting the range of values the variable will take on. This was a challenging task even for the original author of some simple turtle drawing programs. To expect the programmer to know the range of a variable is in a sense begging the question: if the programmer knew its variance, there might be no point to auralizing. This problem is not unknown to the field of software visualization in which limits are imposed by the boundary of screen coordinates. Our next version of LogoMedia will monitor the range of values during each execution of a program and present this information to the user to help guide the next iteration of the auralization specification.

Specifying the monitoring of execution events

In order to provide LogoMedia with simple non-invasive techniques for auralizing control flow, we are developing a method for associating audio commands called audiopoints with individual lines of code. Figure 6 illustrates a sample LogoMedia editor window with the Logo code on the right and its associated audiopoints on the left. To add an audiopoint to a line of the program the user simply positions the cursor in the left column beside the desired line and types a Logo statement. Alternately, the programmer after moving the cursor over the appropriate column can simply play a note on an attached MIDI keyboard. The corresponding LogoMedia command for reproducing this note will then appear as an audiopoint. As the program is running LogoMedia will execute each audiopoint immediately prior to the line of code on its right.



```
                         to factorial :num
○ startnote :click 60 60   if :num = 0
                             [output 1]
● drumup                     [output (:num * factorial (:num - 1))]
● drumdown               end
```

Figure 6: *LogoMedia's two column format for associating audio with program events*

A simple audiopoint such as on the second line can generate a fixed length tone as the code is executed to indicate the line has been reached. Additionally, the programmer can call his or her own custom auralization procedures or choose from a library of useful sound procedures. We are constructing such a library with the ability to toggle sounds on and off and layer sounds. The audiopoint on line 4, for instance, causes drumming sounds to become increasingly dense as the factorial program descends into successively deeper levels of recursion. As the program pops out of the nested levels, the audiopoint on line 5 causes the sound to become less dense.

Conclusions

Whether sound can be used effectively in programming environments remains to be determined. Testing and evaluation of LogoMedia is planned to ascertain which applications of audio in the program development process are most useful. We do not expect sound to replace either traditional textual representations or software visualizations, but to complement them. Further research is needed to determine how all three modalities can be best synthesized to take advantage of the multi-media capabilities of today's computers. We hope that by enhancing the development environment with sound we can make programming more engaging and empower the programmer with tools for more accurate entering, faster debugging, and an improved understanding of a program's function, structure, and behavior.

Acknowledgements

We appreciate the support to our laboratory from the Natural Sciences and Engineering Research Council of Canada, the Information Technology Research Centre of Excellence of Ontario, and Apple Computer, Inc.

References

Baecker, Ronald M. (1981). *Sorting Out Sorting*. Dynamic Graphics Project, Computer Systems Research Institute, University of Toronto. 16 mm color sound film, 30 minutes, distributed by Morgan Kaufman Publishers.

Baecker, Ronald M., and Aaron Marcus (1990). *Human Factors and Typography for More Readable Programs*. Reading, Massachusetts: Addison-Wesley.

Baecker, Ronald M., and J.W. Buchanan (1990). A Programmer's Interface: A Visually Enhanced and Animated Programming Environment. *Proceedings of the Twenty-Third Annual Hawaii International Conference on Systems Sciences*, 531-540.

Blattner, M., D. Sumikawa, and R. Greenberg (1989). Earcons and icons: Their structure and common design principles. HCI 4 (1): 23-37.

Bly, Sara (1982). Presenting information in sound. *Proceedings of the CHI '82 Conference on Human Factors in Computer Systems*, 371-375.

Brown, Marc H. (1988). Exploring Algorithms Using Balsa II. *IEEE Computer* 21 (5): 14-36.

Brown, Marc H., and John Hershberger (1991). *Color and Sound in Algorithm Animation*. Digital Equipment Corporation. Systems Research Center Report, 76a.

Buxton, William,William Gaver, and Sara Bly (in preparation). *Auditory Interfaces: The Use of Nonspeech Audio at the Interface*. Cambridge University Press.

Eisenstadt, M., and M. Brayshaw (1988). The Transparent Prolog Machine (TPM): An Execution Model and Graphical Debugger for Logic Programming. *Journal of Logic Programming* 5 (4): 1-66.

Gaver, W. W. (1989). The Sonic Finder: An interface that uses auditory icons. *HCI* 4 (1): 67-94.

Gaver, W. W. (1990). Auditory icons in large-scale collaborative environments. *Proceedings of the Human-Computer Interaction - Interact '90 Conference*,

Gaver, W. W. (1991a). Effective sounds in complex systems: The ARKola simulation. *Proceedings of the ACM Special Interest Group in Computer-Human Interaction Conference*, 85-90.

Gaver, W. W. (1991b). Sound Support for Collaboration. *Proceedings of the Second European Conference on Computer-Supported Cooperative Work*, 293-308.

Gellenbeck, Edward M., and Curtis R. Cook (1991). Does Signaling Help Professional Programmers Read and Understand Computer Programs? *Proceedings of the Empirical Studies of Programmers: Fourth Workshop*, 82-98.

Harvey, B. (1986). *Computer Science Logo Style: Intermediate Programming*. Cambridge: MIT Press.

Henry, R. R.,K.M. Whaley, and B. Forstall (1990). The University of Washington Illustrating Compiler. *Proceedings of the ACM SIGPLAN '90 Conference on Programming Language Design and Implementation*, 223-233.

Hutchins, E. L., J. D. Hollan, and D. A. Norman (1986). Direct manipulation interfaces. In *User centered system design: New perspectives on human-computer interaction*. Edited by D. A. Norman and S. W. Draper. 87-124. Hillsdale, NJ: Lawrence Erlbaum Associates, Inc.

IMA (1983). *MIDI musical instrument digital interface specification 1.0*. North Hollywood, CA: IMA. (Available from IMA, 11857 Hartsook Street, North Hollywood, CA, 91607, USA)

Jackson, Jay Alan, and Joan M. Francioni (1992). Aural Signatures of Parallel Programs. *Proceedings of the Twenty-Fifth Hawaii International Conference on System Sciences*, 218-229.

Jenkins, James J. (1985). Acoustic information for objects, places and events. *Proceedings of the First International Conference on Event Perception.*,

Kesler, T. E., and R. B. Uram (1984). The effect of indentation on program comprehension. 21 415-428.

Mezrich, J. J.,S. Frysinger, and R. Slivjanovski (1984). Dynamic representation of multivariate time series data. *Journal of the American Statistical Association* 79: 34-40.

Nanja, Murthi, and Curtis R. Cook (1987). An Analysis of the On-Line Debugging Process. *Proceedings of the Empirical Studies of Programmers: Second Workshop*, 172-184.

Patterson, R. D. (1989). Guidelines of the design of auditory warning sounds. *Proceedings of the Institute of Acoustics 1989 Spring Conference*, 17-24.

Price, Blaine A.,Ian S. Small, and Ronald M. Baecker (1992). A Taxonomy of Software Visualization. *Proceedings of the Twenty-Fifth Hawaii International Conference on System Sciences*, 597-606.

Small, Ian S. (1989). Program Visualization: Static Typographic Visualization in an Interactive Environment. Masters Thesis, Department of Computer Science, University of Toronto.

Smith, Ben (1991). Unix Goes Indigo. *Byte* 16 (9): 40-41.

Smith, Stuart,R. Daniel Bergeron, and Georges G. Grinstein (1990). Stereophonic and Surface Sound Generation for Exploratory Data Analysis. *Proceedings of the CHI'90, ACM Conference on Human Factors of Computing Systems*, 125-132.

Sonnenwald, Diane H.,B. Gopinath,Gary O. Haberman,William M. Keese III, and John S. Meyers (1990). InfoSound: An Audio Aid to Program Comprehension. *Proceedings of the Twenty-Third Hawaii International Conference on System Sciences*, 541-546.

Speeth, R.D. (1961). Seismometer sounds. *The Journal of the Acoustical Society of America* 33 (7): 909-916.

Weinberg, Gerald (1971). *Psychology of Computer Programming*. New York: Van Nostrand Reinhold Company.

Yager, Tom (1991). The Littlest SPARC. *Byte* 16 (2): 169-174.

Yeung, E.S. (1980). Pattern recognition by audio representation of multivariate analytical data. *Analytical Chemistry* 52 (7): 1120-1123.

A Framework for Describing and Implementing Software Visualization Systems

John Domingue
Blaine A. Price[1]
Marc Eisenstadt
Human Cognition Research Laboratory
The Open University
Milton Keynes, UK, MK7 6AA
Phone: +44 908 65-3800 (Fax: -3169)
Internet e-mail: `j.b.domingue@open.ac.uk`

Abstract

In recent years many prototype systems have been developed for graphically visualizing program execution in an attempt to create a better interface between software engineers and their programs. Several classification-based taxonomies have been proposed to describe computer program visualization systems and general frameworks have been suggested for implementation. In this paper we provide a framework for both describing existing systems and implementing new ones. We demonstrate the utility of automatic program visualization by re-implementing a number of classic systems using this framework.

Résumé

Récemment on a développé beaucoup de systèmes prototype pour visualiser graphiquement l'exécution d'un programme afin de créer une meilleure interface entre les créateurs de logiciel et leurs programmes. Plusieurs taxonomies basées sur une classification ont été proposées pour décrire des systèmes de visualization de programmes et des cadres généraux ont été proposés pour leur réalisation. Dans ce papier nous fournissons un cadre dans lequel on peut décrire des sytèmes actuels et exécuter de nouveaux systèmes. On montre l'utilité de visualization automatique de programmes en ré-exécutant quelques anciens systèmes dans le cadre de ce prototype.

Keywords: Program Visualization, Algorithm Animation, CASE, Debugging Aids, Software Visualization, Software Engineering

[1]Also affiliated with: The Dynamic Graphics Project, Computer Systems Research Institute, The University of Toronto, Toronto, Canada M5S 1A1

Introduction: What is Software Visualization?

The tools traditionally used by software engineers to help monitor and analyse program execution have been plain ASCII-text based debugging environments which usually allow the user to trace the currently executing code, stop and start execution at arbitrary points, and examine the contents of data structures. Although they can be understood by experts, these tools have a limited pedagogic value and by the early 1980's the work of Baecker and Sherman (1981) and Brown (1988) showed how algorithms could be animated with cartoon-like displays that show a high level abstraction of a program's code and data (Brown referred to this as "algorithm animation").

A concurrent development in the mid-1980's was the appearance of systems which displayed graphical representations that were more tightly coupled with a program's code or data and showed more or less faithful representations of the code as it was executing. Although the displays were not as rich as the custom built algorithm animations, these systems were closer to the tools that software engineers might use. These "program animators" together with the algorithm animators became known as "program visualization" systems. We prefer the more descriptive term "software visualization" (Price, Small, & Baecker, 1992) which encompasses both algorithm and program visualization as well as the visualization of multi-program software systems. In this paper we will use the term software visualization (SV) to describe systems that use visual (and other) media to enhance one programmer's understanding of another's work (or his own).

Classifying SV Systems

One of the first taxonomies of SV was that of Myers (1986) (updated later as (Myers, 1990)), which served to differentiate SV, visual programming, and programming-by-example. In classifying SV systems, Myers used only two dimensions: static vs. dynamic and code vs. data. The first dimension is based on the style of implementation;

static displays show one or more motionless graphics representing the state of the program at a particular point in time while animated displays show an image which changes as the program executes. The second dimension describes the type of data being visualized, be it the program source code or its data structures.

The taxonomies that have been proposed since Myers have also used few dimensions, which seems to ignore that fact that there are many styles for implementation and interaction as well as different machine architectures and ways of utilizing them. Price, Small, and Baecker (1992) recently proposed a taxonomy which describes 6 broad categories for classifying SV systems: scope, content, form, method, interaction, and effectiveness. Each of these categories has between three and seven characteristics, for a total of thirty dimensions to describe each system. This pragmatic classification system provides a means for comparing the functionality and performance of a wide range of SV systems, but it does not provide a language or framework for implementing new systems. Eisenstadt et al. (1990) described nine qualitative dimensions of visual computing environments which can form the basis of a language for describing SV systems, but these serve only to describe the attributes of systems rather than drive their construction.

From Taxonomy to Framework and System: "what goes on?"

Taxonomies are useful, but we need more if we are to provide a firm basis upon which to describe SV systems in depth, let alone implement them. A *framework* for describing SV systems could provide extra leverage by being a little more prescriptive, i.e. making a commitment regarding how to approach the design and construction of SV systems. In fact, it is not a very big step from specifying such a framework to designing a system for *building* a SV system (SV system-building system). An important difference is that the former activity is merely a paper exercise, whereas the latter activity is intended to lead towards a working tool. Indeed, the latter activity serves as a useful forcing function: it encourages us to build re-usable libraries of software that we believe encapsulate important generalizations about SV system-building. The proof of the soundness of a design built in this way lies in the ability to use it both to reverse-engineer existing SV systems and construct new systems with ease.

Programming Language Visualization vs Algorithm Animation

Several of the noteworthy SV building systems and frameworks focus on algorithm animation, which means that the animations that they produce are custom designed and each new program requires manual annotation to animate it. Programs are animated in BALSA (Brown, 1988) by adding calls to the animation system at "interesting events" in the code. Systems implemented by Stasko (1990) and London and Duisberg (1985) provide facilities for smooth transitions in animations based on "interesting event" calls.

In our work, we have focussed on supporting the construction of systems which visualize the execution of programming languages. By visualizing a programming language interpreter (or compiler) one also, in some sense, automatically gets a visualization for any program written in that language (thus achieving the *automatic* goal suggested by Price, Small, & Baecker (1992)). Programming language visualization (PLV) and algorithm animation (AA) overlap, but there are differences in the approach. AA systems typically show a very high level picture of a program's execution and the images that it generates can be far removed from the data structures contained in the program. The animations cover a narrow set of programs (typically a single algorithm). PLVs on the other hand have to deal with any program which can be realised in the language. Thus PLV displays usually have much simpler images than AA displays since they must be highly generalized whereas AA displays can be custom tuned. The problem for an AA system is to show the characteristics (signature) of an algorithm as clearly as possible. The problem for a PLV is to allow arbitrarily large execution spaces to be examined in a comprehensible fashion.

Our approach is to concentrate primarily on PLV, but to provide generalizations which are applicable to AA as well. In the rest of this paper we describe the design of a SV system-building system (and framework) called Viz, which we have implemented as a prototype running in Common Lisp and CLOS on Sun workstations. Our Viz implementation has already been used to reconstruct three well known PLV systems: an OPS-5 visualization system (based on TRI (Domingue & Eisenstadt, 1989)), a Prolog visualization system (based on TPM (Eisenstadt & Brayshaw, 1988)), and a Lisp tracer (based on the Symbolics™ tracer). After describing the Viz architecture, we explain how one of these reconstructions was implemented. In order to explore the relationship between Viz's PLV-oriented approach and AA-oriented systems we have also used Viz to implement some of the animations from Brown's BALSA (sorting) and Stasko's TANGO (bin-packing). We conclude with a comparison of the terminology used in BALSA, TANGO, and Viz to describe abstractions and we highlight aspects of the Viz design.

Viz Architecture

In Viz, we consider program execution to be a series of *history events* happening to (or perpetrated by) *players*. To allow SV system builders considerable freedom, a player can be any part of a program, such as a function, a data structure, or a line of code. Each player has a name and is in some *state*, which may change when a history event occurs for that player. A player may also contain other players, enabling groups of players to be formed. History events are like Brown's "interesting events" in BALSA—each event corresponds to some code being executed in the

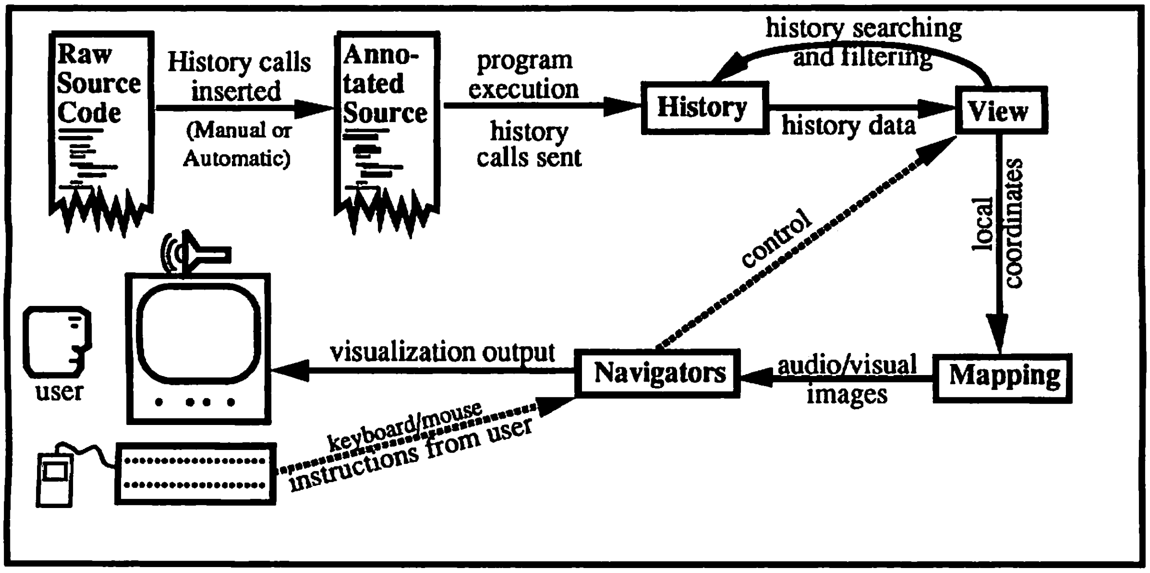


Figure 1. The Architecture of Viz

program or some data changing its value. These events are recorded in the history module, which allows them to be accessed by the user and "replayed." Events and states are *mapped* into a visual representation which is accessible to the end-user (the programmers who need to use the SV system, not the SV system builder). But the mapping is not just a question of storing pixel patterns to correspond to different events and states—we also need to specify different views, and ways of navigating around them. The main ingredients of Viz are:

- Histories: a record of key events that occur over time as the program runs, with each event belonging to a player; each event is linked to some part of the code and may cause a player to change its state (there is also some pre-history information available before the program begins running, such as the static program source code hierarchy and initial player states).
- Views: the style in which a particular set of players, states or events is presented, such as using text, a tree, or a plotted graph; each view uses its own style and emphasizes a particular dimension of the data that it is displaying.
- Mappings: the encodings used by a player to show its state changes in diagrammatic or textual form on a view using some kind of graphical language, typography, or sound; some of a player's mappings may be for the exclusive use of its navigators.
- Navigators: the tools or techniques making up the interface that allows the user to traverse a view, move between multiple views, change scale, compress or expand objects, and move forward or backward in time through the histories.

This framework is equally at home dealing with either program code or algorithms, since a player and its history events may represent anything from a low-level (program code) abstraction such as "invoke a function call" to a high level (algorithm) abstraction such as "insert a pointer into a hash table."

Figure 1 shows the general architecture of Viz. The target system source code is annotated to generate history calls. When the system being visualized is a programming language, hooks into the interpreter or compiler are used to generate history events. As the code executes, the inserted calls cause "interesting events" regarding players to be recorded in the history module.

When the user runs the visualization, the view module reads the history data at the request of the navigator. The view module sets the layout of the history events and sends local coordinates for each history datum through the mapping module, which draws a graphical or textual representation for each event. The screen images are then transformed and presented on the screen by the navigator. The user interacts with the visualization using the navigator, which sends control signals to the view module to cause all changes in the visualization, such as panning, zooming, local compression and expansion, and moving forward and backward in time through the program execution space.

Histories

The first task for the visualization programmer using Viz is to decide what types of events may occur during program execution, which elements in the program will be the players and how the players change state. After defining these, the programmer may insert *create new player* and

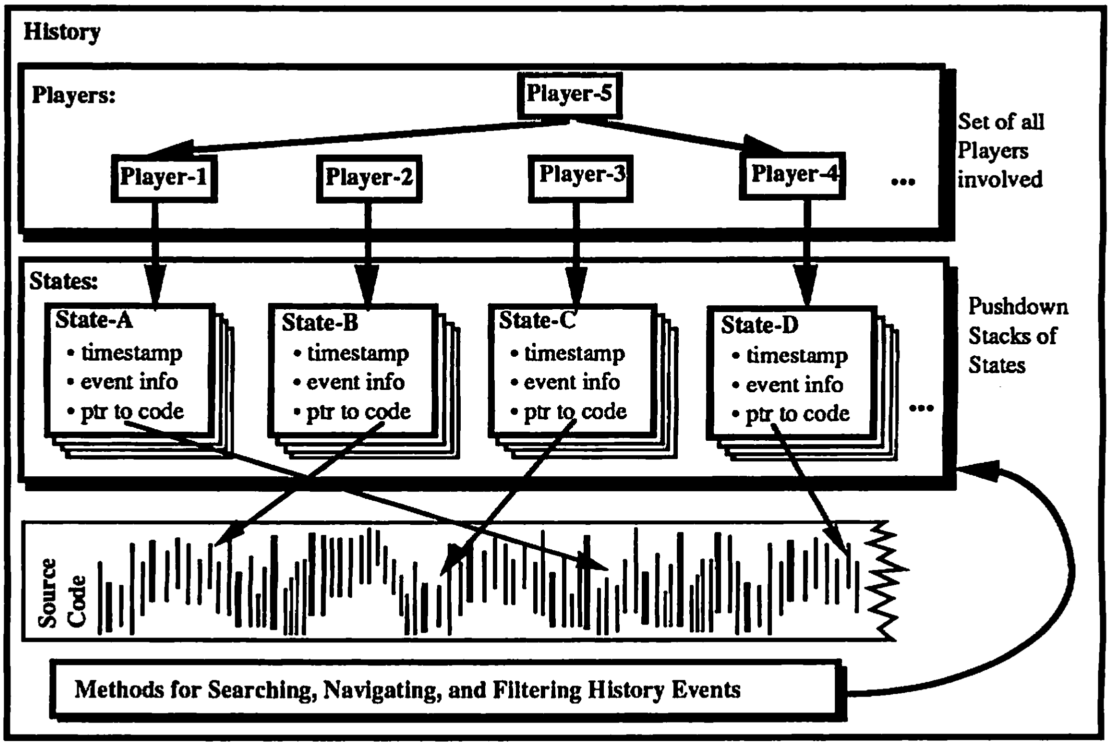


Figure 2. A Prototypical history structure.

note event calls in the code, which form the interface between a program and its visualization.

Figure 2 shows a prototypical history structure in the history module. This consists of a set of players and a sequence of history states. Each player has a name, a pointer to its current history state and a pushdown stack of previous states. A player may contain other players, as shown by player 5 in figure 2. This feature is useful in navigation. Each state has a timestamp, a pointer to the appropriate segment of source code and an event structure. As a program executes, new players and history states are created, and existing players are "moved" into new states, pushing previous states onto a stack. The various states of the players are caused by the different types of events.

The choice of players and event types together with the judicious placement of *note event* calls in code determine the execution model. Currently, we do not advocate any methodology for creating the execution model, except to point out that events are the "things that happen" in a program causing a player or players to move into a particular state.

Views

A view can be thought of as a perspective or window on some aspect of a program or algorithm, with (possibly) many views making up a visualization. There are some similarities between our views and the animation views, adapted from the Model-View-Controller paradigm, described by London and Duisberg (1985). The main difference is that within the animation views, the layout, handled by views in Viz, and appearance, handled by mappings in Viz, are handled together. Each view in Viz can be thought of as embodying a style of formatting collections of objects. Within Viz we have constructed a hierarchy of views, including text, graph, table and tree based views, each embodying a particular layout style, which can be used or specialized by the SV builder.

A view requests data from the history module and sends it to the mapping module, which decides the appearance. The view module then tells the mapping module where to display the mapped data. This means that the view module is concerned only with the position of the history data item, not its appearance.

The view module is also responsible for managing the compression (ellision) and expansion of elements in the display based on user commands from the navigator. If the user selects the compression of display elements, such as a subtree in a tree hierarchy, then the view module groups the players concerned into a new player which has its own mapping to represent the compressed players. When the navigator tells the view module that an element is to be expanded, the view disbands that new player and displays the individual mappings for the players.

	Prolog	OPS5	Lisp	Sort	Bin-Packing
Player	predicate instantiation (goal)	rule	form	data item	data item bin

States	pending goal;
	succeeded;
	failed;
	failed on backtracking;
	redo-goal
		failing to match working memory;
		firing
			unevaluated;
			evaluated
				location	attempting to fit;
					suceeded; new

Events	call;
	exit;
	fail-1st;
	fail-nth;
	redo
		choose for firing	call;
			return
				assignment of item to cell	attempt-fit
					succeed-fit
					new-item

Mappings	▨; ▢; ■; ▨; ▨ (colour)	blank; ✚	-> *italic*; <- **bold**	●	▯; ▮; ▨
Views (in order of decreasing granularity)	tree: players, players current state; formatted text	table: players vs. cycles, player's state @ cycle formatted text	pretty printed code: player's current state	point plot: players, player's value & current state; formatted text	point plot using rectangles: bin-players and current state

Table 1: A Viz description of five example systems

Mappings

The goal of a mapping in Viz is to communicate the maximum amount of information about a player's state while imposing the least possible cognitive load on the user.

In conjecturing a theory of effectiveness of graphical languages, Mackinlay (1986) noted Cleveland and McGill's observation that people accomplish the perceptual tasks associated with the interpretation of graphical presentations with different degrees of accuracy. Using psychophysical results, Mackinlay extended Cleveland and McGill's work to show how different graphical techniques ranked in perceptual effectiveness for encoding quantitative, ordinal, and nominal data. He found that the position of the data item in the x-y plane is ranked first for all three types of data, which is why we separate the view layout from the mappings. The other techniques which may be varied to create an effective mapping are (in decreasing order of effectiveness): colour hue, texture, connection, containment, density (brightness), colour saturation, shape, length (size), angle or slope (orientation), and area (or volume).

A mapping in Viz is attached to a particular type of player, event or state, and view. Multi-method inheritance occurs over the class of entity and view, allowing a Viz user to formulate expressions such as "all entities in view-x are to be displayed as a filled triangle", "entity-y is always displayed as a white circle" and "entity-a is displayed as a circle in tree based views but as a square in all other views". Mappings can be inherited, forming an inheritance hierarchy in much the same fashion as views. Our future work in Viz will create a library of mappings.

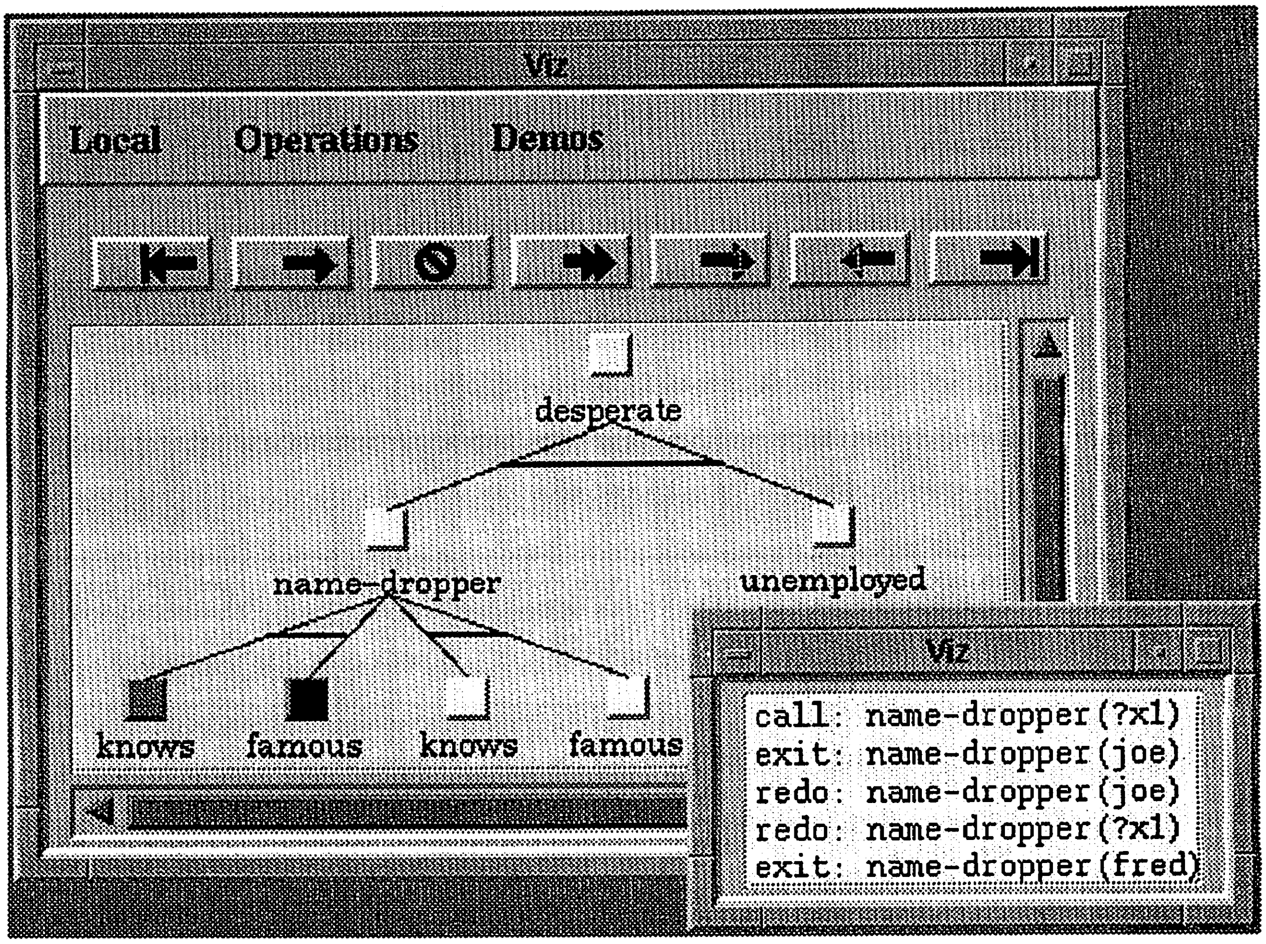
Navigators

The Viz navigator module encapsulates the interface between the user and the visualization, although the methods for performing the navigation tasks are found in the view module, thus allowing custom navigation interfaces to be built independently of the task.

Our prototype provides a replay panel (see the screen snapshot in figure 3) for searching, which has buttons for moving to the beginning or end of the animation, single-stepping forward or backward, playing forward, fast-forwarding and stopping. Stepping in Viz involves notifying the history and view modules of the change of focus (the history module then selects the next appropriate event). Horizontal and vertical scroll bars are provided for panning while simple zoom in and zoom out buttons provide scaling. The user can select a fine grained view of a data element by clicking on it.

Examples Defined in Viz

The descriptions of the three systems that we have implemented using Viz are presented in table 1 along with the two examples from BALSA and TANGO animations. The table provides a summary of the players, states, events, mappings, and views used in each visualization. Each row represents a distinct Viz entity type and each column represents one of the visualizations. The player row lists the players which can take part in each example. The states



```
call: name-dropper(?x1)
exit: name-dropper(joe)
redo: name-dropper(joe)
redo: name-dropper(?x1)
exit: name-dropper(fred)
```

Figure 3: Screen Snapshot of Prolog Visualizer (greyscale from a colour screen)

row shows the possible states players can enter. The events row shows the events which cause state changes. The mappings row contains, in order, the icon mapping for each state. The views row lists the names of the possible views in decreasing order of granularity. The connection between a view and the history is also shown. We shall now explain the first column, the Prolog visualizer, in detail.

A Prolog Visualizer

The Prolog visualizer is based on the Transparent Prolog Machine (TPM) (Eisenstadt & Brayshaw, 1988). TPM uses an AND-OR tree representation where the nodes represent goals which are instantiated Prolog predicates, and the arcs represent conjunctions or disjunctions of subgoals.

The players in the visualization are the instantiated Prolog predicates or goals in the proof tree. The events, which are adapted from the Byrd Box Model (Byrd, 1980), are: call (trying to prove a goal), exit (a goal succeeding), fail-1st (a goal failing the first time attempted), fail-nth (a goal having succeeded earlier, later failing on backtracking), and redo (re-attempting to satisfy a goal). There is a corresponding state and mapping for each event type (shown in respective order so ▨ equals call, □ equals exit, etc.).

The Prolog interpreter takes a list of goals left to prove. When no goals are left the environment is returned. The algorithm (adapted from (Nilsson, 1984)) for the interpreter is:

```
If there is nothing to prove
then return the environment; else
    if the first-goal-left-to-prove
    is true, then
        note event: succeed, the goal that
                    was proved, env,
                    and prove the
                    remaining goals; else
        note event: goal,
                first-goal-left-to-prove,
                env, and create a player for
                the-first-goal-left-to-prove,
                and
        loop for each clause in the database
                if the head of the clause
                matches the first goal then
```

```
                    create a player for
                    each of the subgoals
                    in the clause
                if we prove the new list of
                    goals (which is the
                    body of the matched
                    clause appended to the
                    rest of the goals)
                    then return the new
                    environment else
                note event: redo,
                    first-goal-left-to-prove, env,
            note event: failure,
                first-goal-left-to-prove, env,
                and
            return failed-to-prove
                first-goal-left-to-prove.
```

In the above, the algorithm is shown in italics while the Viz event calls are shown in plain text.

Figure 3 shows a screen snapshot of the proof for the goal ?- desperate(?x) given the following Prolog database:

```
desperate(?x) :-
  name-dropper(?x),
  unemployed(?x).

name-dropper(?x) :-  knows(?x, ?y),
                     famous(?y).

name-dropper(?x) :-  knows(?y, ?x),
                     famous(?y).

knows(joe, mick).
knows(charles, fred).
famous(mick).
famous(charles).
unemployed(fred).
```

Bearing in mind that atoms beginning with "?" depict variables in this approximation of Edinburgh-syntax Prolog, the first five lines of the code above defines a) the relation that someone is desperate if they are name dropper and unemployed; and b) a person is a name dropper if they know someone who is famous or if someone famous knows them. The rest of the code defines five "facts" about who knows who, who is famous and who is unemployed.

Because history events are invoked by inserting hooks into our own Prolog interpreter, the Viz implementation is straightforward once the machinery for players, views, mappings, and navigators is in place. The simple implementation, as described here, can deal with non-trivial cases of tricky backtracking and unification. Coping with arbitrarily large proof trees requires the definition of a "collapsed predicate" player. The tree beneath a collapsed predicate player would not be displayed unless requested (by clicking on it with the mouse). The current set of collapsed predicates would be chosen by the user. This collapsed predicate set would correspond to the segments of code the user deemed irrelevant and thus could be "black boxed away". In order for collapsed predicates to be distinguishable, the mapping for a collapsed predicate player would be a triangle. The collapsed predicate player would in fact contain the players within its subtree. A request to show the full sub-tree would result in the replacement of the single collapsed predicate player with the players it contained.

BALSA	TANGO	Viz	Comments
Interesting (Algorithm) Events	Algorithm Operations	Events and Create Players	The BALSA and TANGO terms are virtually identical while Viz events can be arranged hierarchically, and are designed to relate to the code rather than the algorithm.
Modellers	Image, Location, Path, and Transition	States and Players	In describing a visualization's internal representation, TANGO adds to the BALSA framework by providing 4 abstract data types (geared towards animation); Viz's states and players are program execution level abstractions.
Renderers	Animation Scenes	Mappings and Views	BALSA provides a general mechanism for each view while TANGO provides reusable libraries of animation scenes; Viz discriminates between the actual images that are mapped to the screen and the style in which they are displayed (the view).
		Navigators	BALSA and TANGO don't specify any kind of user interface interactions within the framework, nor techniques for dealing with arbitrarily large programs.
Adaptor and Update Messages		History	The Viz history is a structure for the collection of events, states, and players generated during program execution. The history module includes various searching and filtering functions.

Table 2: A Comparison of Terminology

Additional Systems

To fully exercise our evolving framework across a range of players, events, states, mappings, and views, we have also used Viz to re-implement the textual visualization provided by the Symbolics™ Lisp stepper/tracer which uses layout to summarize the execution history of the Lisp evaluator (the Viz implementation actually improves on this by using colour and typography as well) and the table based vizualization of an OPS-5 style rule interpreter. We have also duplicated some of the well known AA examples from BALSA (sorting) and TANGO (bin packing) to show that the system can be used to easily construct custom algorithm animations as well.

A Comparison of Viz, Balsa, and Tango Terminology

Although each of the goals of Viz, BALSA, and TANGO are somewhat different, the importance of the pioneering work of BALSA and TANGO is such that a close comparison of terminology is warranted. Viz terminology is designed to allow existing systems to be described as well as implement new ones. Table 2 shows the systems in left-to-right chronological order, mapping the similar terminology across systems where appropriate, and highlighting differences accordingly in the "comment" column.

Conclusions

The main goal in designing Viz was to provide a descriptive mechanism for understanding and explaining the diverse notations and methodologies underlying existing software visualization environments. Our re-implementation based approach is in contrast to the current literature (Eisenstadt et al., 1990; Green, 1989; Green, 1990; Myers, 1990; Price et al., 1992) which focusses on cognitive and notational dimensions and practical categories. The amount of effort involved in our approach is of the same order of magnitude as category or dimension based approaches. Each of the example systems was constructed within 1-2 days (this included many extensive alterations to the first version of Viz) and is of the order of 100 lines of code.

By providing a descriptive abstraction for internally expressing the state of an algorithm (players, events, and states) we have augmented earlier frameworks and added to the common language for representing algorithm animation designs. Since Viz provides visualization facilities for programming languages, we have provided a framework for generalized visualization that is applicable to software engineers since it provides visualizations that are automatic and faithful to the execution model of the language. The use of Viz to implement systems which differ widely in terms of their scope, content, form, method, interaction, and effectiveness, suggests that the framework is sufficient to design and implement a wide class of software visualization systems.

Acknowledgements: This research was supported by CEC ESPRIT-II Project 5365 (VITAL), the UK SERC/ESRC/MRC Joint Council Initiative on Cognitive Science and Human Computer Interaction Project 91/CS66, and NSERC. We wish to thank Ron Baecker, Mike Brayshaw, Thomas Green, and Marc Brown for their helpful comments on early drafts of this paper.

REFERENCES

Baecker, R. M. & Sherman, D. (1981). *Sorting Out Sorting*. narrated colour videotape, 30 minutes, presented at ACM SIGGRAPH '81. Los Altos, CA: Morgan Kaufmann.

Brown, M. H. (1988). *Algorithm Animation*. New York: MIT Press.

Byrd, L. (1980). Understanding the Control Flow of Prolog Programs. In S. A. Tarnlund (Ed.), *The 1980 Logic Programming Workshop*, (pp. 127-138).

Domingue, J. & Eisenstadt, M. (1989). A New Metaphor for the Graphical Explanation of Forward Chaining Rule Execution. In *The Eleventh International Joint Conference on Artificial Intelligence*, (pp. 129-134).

Eisenstadt, M. & Brayshaw, M. (1988). The Transparent Prolog Machine (TPM): an execution model and graphical debugger for logic programming. *J. of Logic Prog.*, 5(4), 1-66.

Eisenstadt, M., Domingue, J., Rajan, T., & Motta, E. (1990). Visual Knowledge Engineering. *IEEE Trans. on Software Engineering*, 16(10), 1164-1177.

Green, T. R. G. (1989). Cognitive Dimensions of Notations. In A. Sutcliffe & L. Macaulay (Eds.), *People and Computers V* (pp. 443-460). Cambridge: Cambridge University Press.

Green, T. R. G. (1990). The Cognitive Dimension of Viscosity: a sticky problem for HCI. In D. Diaper, D. Gilmore, G. Cokton, & B. Shackel (Ed.), *INTERACT '90 Conference on Computer-Human Interaction*, (pp. 79-86). Amsterdam: Elsevier.

London, R. L. & Duisberg, R. A. (1985). Animating Programs using Smalltalk. *IEEE Computer*, **18**(8), 61-71.

Mackinlay, J. (1986). Automating the Design of Graphical Presentations of Relational Information. *ACM TOGS*, 5(2), 110-141.

Myers, B. A. (1986). Visual Programming, Programming by Example, and Program Visualization: A Taxonomy. In M. Mantei & P. Orbeton (Ed.), *CHI '86 Human Factors in Computing Systems*, (pp. 59-66). New York: ACM.

Myers, B. A. (1990). Taxonomies of Visual Programming and Program Visualization. *JVLC*, 1(1), 97-123.

Nilsson, M. (1984). The world's shortest Prolog interpreter? In J. A. Campbell (Eds.), *Implementations of Prolog* (pp. 87-92). Chichester, England: Ellis Horwood.

Price, B. A., Small, I. S., & Baecker, R. M. (1992). A Taxonomy of Software Visualization. In *The 25th Hawaii International Conference on System Sciences*, Volume II (pp.597-606). New York: IEEE.

Stasko, J. T. (1990). The Path-transition Paradigm: a practical methodology for adding animation to Program Interfaces. *J. of Vis. Lang. and Comp.*, 1(3), 213-236.

Volume Rendering using the Fourier Projection-Slice Theorem

Marc Levoy

Computer Science Department
Center for Integrated Systems
Stanford University
Stanford, CA 94305-4070
Email: levoy@cs.stanford.edu

Abstract

The Fourier projection-slice theorem states that the inverse transform of a slice extracted from the frequency domain representation of a volume yields a projection of the volume in a direction perpendicular to the slice. This theorem allows the generation of attenuation-only renderings of volume data in $O(N^2 \log N)$ time for a volume of size N^3. In this paper, we show how more realistic renderings can be generated using a class of shading models whose terms are Fourier projections. Models are derived for rendering depth cueing by linear attenuation of variable energy emitters and for rendering directional shading by Lambertian reflection with hemispherical illumination. While the resulting images do not exhibit the occlusion that is characteristic of conventional volume rendering, they provide sufficient depth and shape cues to give a strong illusion that occlusion exists.

Keywords: Volume rendering, Fourier projections, Shading models, Scientific visualization, Medical imaging

1. Introduction

Volume rendering is a technique for visualizing a sampled scalar or vector volume of three spatial dimensions by modeling the propagation of light in a participating medium. In its most general form, parameters of the data are mapped to physical parameters of the medium such as energy density, absorption and scattering coefficients, and scattering phase function [Krueger90]. The integro-differential equation governing light transfer is then solved to obtain an energy equilibrium for the volume [Siegal81].

To reduce solution costs, practical volume rendering algorithms assume a low albedo, allowing them to ignore the effects of interreflection [Levoy88, Sabella88, Drebin88]. The resulting equation can be evaluated independently for each viewing ray using an expression of the form

$$I = \int_{-\infty}^{\infty} f(t) \exp\left[-\int_{-\infty}^{t} g(u)\, du\right] dt \qquad (1.1)$$

where t represents distance along the ray. This expression is typically evaluated by digital compositing [Porter84], which in the context of volume rendering is equivalent to numerical integration using the rectangle rule. Given a cubical volume measuring N voxels on a side, and assuming pixel spacing equal to voxel spacing, the cost of generating an image using these algorithms is $O(N^3)$.

If we simplify the physical model still further by ignoring emission and first scatter, we produce an image that looks like an X-ray. The expression for each viewing ray now takes the form

$$I = \exp\left[-\int_{-\infty}^{\infty} g(t)\, dt\right]. \qquad (1.2)$$

For an imaging geometry that employs uniformly spaced parallel rays, expressions of this form can be evaluated efficiently using the Fourier projection-slice theorem. This theorem allows us to compute integrals over volumes by extracting slices from a frequency domain representation of the volume. The application of this theorem to image synthesis has been independently proposed in [Dunne90] and [Malzbender]. An application to MR angiography is described in [Napel91].

The computational advantage of the projection-slice theorem is, at least in theory, considerable. Given a cubical volume and pixel spacing equal to voxel spacing as above, the cost per image is $O(N^2 \log N)$ after an $O(N^3 \log N)$ preprocessing step to compute the 3D frequency domain representation. For a volume and image of width $N = 256$, volume rendering using digital compositing requires $16M$ operations, while rendering using the projection-slice theorem requires only $512K$ operations, a speedup of more than an order of magnitude.

There are two disadvantages to this approach. From a practical standpoint, signal processing considerations have thus far prevented researchers from obtaining speedups matching the theoretical complexity. We expect these problems to be solvable, and, aside from a brief explanation in section 2, we do not address them further in this paper. A more fundamental disadvantage of the Fourier approach is that the resulting images do not exhibit occlusion and are consequently not as useful as volume renderings for many applications. Fortunately, occlusion is only one of many cues employed by the human visual system to determine the shape and spatial relationships of objects. Other cues include perspective, shading, texture, shadows, atmospheric attenuation, stereopsis, binocular convergence, ocular accommodation, head motion parallax, and the kinetic depth effect.

In this paper, we consider techniques for augmenting equation 1.2 to include as many depth and shape cues as possible while retaining the computational efficiency of evaluation using the projection-slice theorem. The key restriction placed on us is that, to avoid recomputing the 3D frequency domain representation for each view, the integrand must be independent of viewing direction. If we wish to support rotation of the volume relative to a light source, the integrand must also be independent of lighting direction. A class of shading models that satisfies these constraints has the form for each viewing ray

$$I = \sum_{i=0}^{n} w_i \left[\int_{-\infty}^{\infty} f_i(t)\, dt \right] \tag{1.3}$$

for scalars w_i and scalar volumes f_i. As we shall see, a surprisingly large class of shading models are expressible as linear combinations of this form where the f_i's are independent of both viewing and lighting directions. In particular, we consider the following three models:

- X-rays with depth cueing
- X-rays with directional shading
- X-rays with depth cueing & directional shading

The remainder of the paper is organized as follows. Section 2 describes the Fourier projection-slice theorem and gives an algorithm for using it to generate arbitrarily oriented X-rays of a sampled scalar volume. In section 3, we derive formulations for the three shading models listed above plus a few extensions. Section 4 describes our implementation and results, and section 5 gives conclusions and suggests directions for future research.

2. The projection-slice theorem

The Fourier projection-slice theorem states that the inverse 2D Fourier transform of a slice passing through the origin of the 3D Fourier transform of a function gives a projected image of the function, where the projection is in a direction perpendicular to the extracted slice. Each point on the projection is the integral of the function over the infinite ray passing through that point and normal to the slice [Dudgeon84]. For later use, we define a Fourier projection operator Π_V with input function f on $\mathbf{R}^3$ and output image I on $\mathbf{R}^2$ such that

$$I = \Pi_V(f) = F_2^{-1}\{F_3\{f\}\,\delta_V\}. \tag{2.1}$$

δ_V restricts the spectrum to a plane passing through the origin and perpendicular to viewing direction V, and F_N and F_N^{-1} are the forward and inverse N-dimensional Fourier transforms, respectively.

The Fourier projection-slice theorem is the basis for one variant of computed tomography (CT). Specifically, if the 2D Fourier transforms of a set of X-rays of an object taken at different angles are inserted into a 3D spectrum such that each slice passes through the origin and is perpendicular to the (parallel) X-ray beam used to acquire that slice, the inverse 3D Fourier transform of the composite spectrum yields a 3D radiometric facsimile of the object, i.e. a CT dataset [Macovski83].

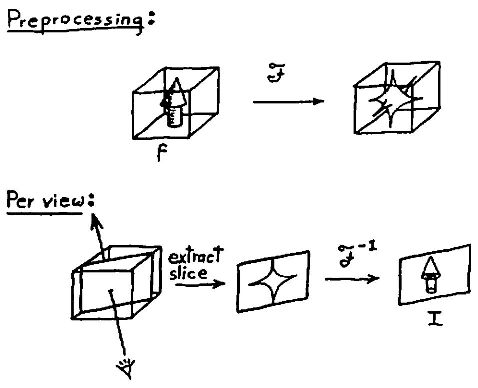


Figure 1: Algorithm for generating an attenuation-only rendering of a volume dataset using the Fourier projection-slice theorem.

By reversing the CT acquisition process, we can generate X-rays from CT data. A typical implementation is shown in figure 1. In a preprocessing step, we compute the 3D discrete Fourier transform (DFT) of a scalar volume sampled on the three-dimensional integer lattice. For real volumes (i.e. having no imaginary component), a discrete Hartley transform (DHT) [Bracewell86] may be substituted. For each view, we extract a slice passing through the origin and compute its inverse 3D DFT. Since an arbitrarily oriented slicing plane does not typically pass through points on the integer lattice, slice values must be computed by interpolation from nearby samples.

For a volume measuring N voxels on a side where N is an integer power of two, the computation of a 3D spectrum using a Fast Fourier Transform (FFT) or Fast Hartley Transform (FHT) algorithm requires $O(N^3 \log N)$ operations. The cost of extracting a 2D slice measuring N samples on a side from the 3D spectrum is $O(W^3 N^2)$ where W is the nonzero width of the interpolating filter. To complete the algorithm, we perform a 2D inverse FFT or FHT, which requires $O(N^2 \log N)$ operations. Properly speaking, sampling theory demands that we employ 2x oversampling during slice extraction. We have found that the penalty of not doing so is minor for the datasets we typically encounter. Assuming $N = 256$ and $W = 4$, we have slice extraction and inverse transformation costs of $16M$ and $2.25M$ operations, respectively. Therefore, although the Fourier transformation is asymptotically dominant, the slice extraction step usually takes more time.

The data structures required by the algorithm (in addition to the input volume) include a 3D spectrum measuring N

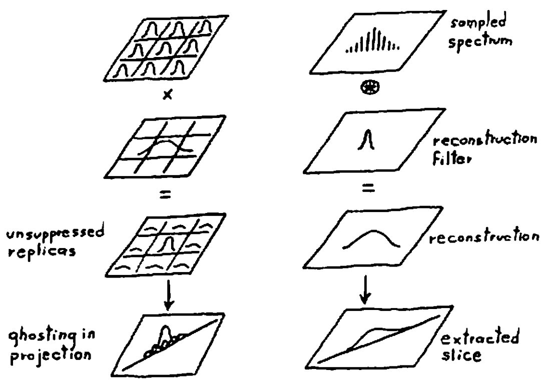


Figure 2: Imperfect reconstruction in the frequency domain produces incompletely suppressed replicas in the spatial domain. These appear in the image as noticeable ghosting.

coefficients on a side and a 2D slice that also measures N coefficients on a side. While the input volume may be adequately represented by 8 bits per voxel, accurate frequency domain representations typically require 32-bit floating point coefficients, thereby increasing memory requirements by a factor of 4. Quantization or coding of Fourier coefficients such as specified in the JPEG still picture compression standard is possible, but reconstruction time must be considered.

The use of finite support interpolation filters in the slice extraction step can give rise to objectionable artifacts in Fourier projections. Although these problems are not the focus of this paper, they impact performance significantly, so a brief discussion of them is warranted.

Convolution of a sampled function with a filter of finite support causes incomplete suppression of periodic replicas in the other domain. Usually, convolution is performed in the spatial domain and the incompletely suppressed replicas occur in the frequency domain. In the Fourier projection algorithm, we convolve in the frequency domain, producing incompletely suppressed replicas in the spatial domain that appear as noticeable ghosting. The problem is summarized in one lower dimension in figure 2. Two solutions are known for this problem:

(1) Employ a better (and typically wider) interpolation filter. If we increase the nonzero filter width by a factor of k_1, $k_1 > 1$, we increase per-view processing time by a factor of $k_1{}^3$.

(2) Widen the separation between spatial domain replicas by zero padding the input volume. If we increase the volume width by a factor of k_2, which corresponds to surrounding the data on each of its six sides by a margin of zeros of width $N(k_2 - 1)/2$, we increase per-view processing time by a factor of $k_2{}^2$. We also increase memory requirements by a factor of $k_2{}^3$ and preprocessing time by a factor of $k_2{}^3\,(log\ k_2 + \log N) / \log N$, but these effects are less important.

The second solution has theoretically superior per-view performance. The relative impact of these two solutions on image quality has not been studied, however. Malzbender reports satisfactory results with a 5^3 tap filter and a 10% margin of zeros on each side of the data [Malzbender]. The images in this paper were made using a 16% margin ($k_2 = 4/3$) and a 4^3 tap filter.

3. Shading models

As discussed in the introduction, the class of integrals that can be directly evaluated using the Fourier projection-slice theorem does not allow the modeling of emission and scattering in a participating medium. In this section, we consider several shading models that are linear combinations of Fourier projections in the spirit of equation 1.3 and which restore some of the lost visual cues.

3.1. Depth cueing

The fraction of light lost to absorption or scattering in a participating medium is exponentially related to the distance it travels. This attenuation provides strong perceptual cues concerning the relative depth of objects in the scene. In this section, we develop an approximate shading model in which exponential attenuation with locally varying absorption and scattering coefficients is replaced by linear attenuation with a constant coefficient. It will be shown that images based on this model can be rendered efficiently using the projection-slice theorem.

Let us begin with exponential attenuation. We define a volume of density $\rho(x,y,z)$ and color $C(x,y,z)$ to emit light with an energy of $C\rho$ per unit length and absorb light with an opacity of $\tau\rho$ per unit length where τ is a constant. We also define a viewing ray $\mathbf{P}(t)$ parameterized by length t and two points $\mathbf{P}(a)$ and $\mathbf{P}(b)$ lying on the ray. Ignoring scattering, the total energy I_P arriving at $\mathbf{P}(b)$ due to emission and absorption along that portion of the ray lying between $\mathbf{P}(a)$ and $\mathbf{P}(b)$ is given by

$$I_P = \int_a^b \rho\,(\mathbf{P}(t))\,C\,(\mathbf{P}(t))\,\exp[-\int_a^t \tau\,\rho\,(\mathbf{P}(u))\,du\,]\,dt. \quad (3.1.1)$$

The notation is adapted from [Max90]. Replacing exponential attenuation with locally varying coefficients $\tau\rho$ by linear attenuation with constant coefficient τ gives

$$I_P = \int_a^b \rho\,(\mathbf{P}(t))\,C\,(\mathbf{P}(t))\,[\tau/2 + \tau/2\,(\mathbf{D}\,(\mathbf{P}(t)) \cdot \mathbf{V})]\,dt \quad (3.1.2)$$

where $\mathbf{D} = (D_x, D_y, D_z)$ is a linear function of position along the ray, i.e. $\mathbf{D} = m\,\mathbf{P} + b$ for constants m and b, and $\mathbf{V} = (V_x, V_y, V_z)$ is the normalized viewing direction as shown in figure 3. Note that $\mathbf{V}$ points away from the eye, not towards it as is common in many textbooks. We have chosen this convention in order to underscore the similarity between equation 3.1.2 and equation 3.2.3 in section 3.2.

In the introduction, we stated that in order to avoid recomputing the 3D frequency domain representation for each

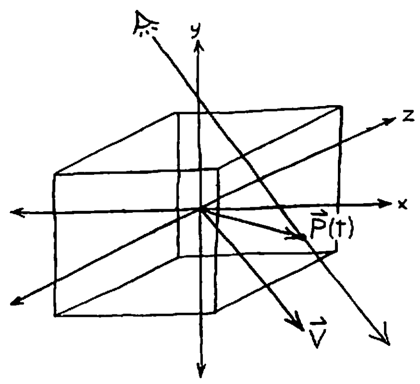


Figure 3: Linear depth cueing for a volume centered at the origin is given by $\mathbf{D} = m\,\mathbf{P} + b$ for constants m and b where $\mathbf{P}(t)$ is position along a viewing ray and $\mathbf{V}$ is the normalized viewing direction.

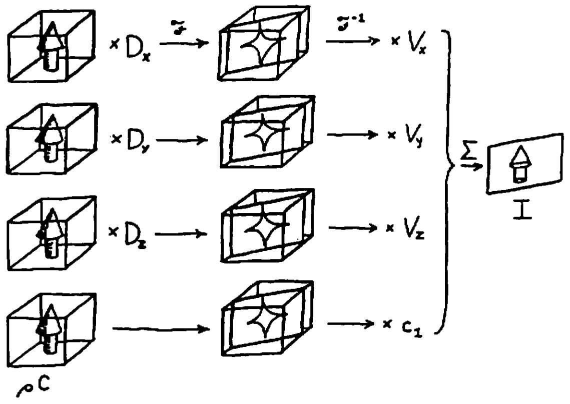


Figure 4: Algorithm for generating a depth cued rendering of a volume dataset. During precomputation, volume $\rho\,C$ is multiplied by depth operator **D**. For each view, the extracted slices are multiplied by viewing direction **V**.

view, all integrals over the volume must be independent of viewing direction. Looking back at the last equation, we observe that viewing direction **V** is independent of ray parameter t. We may therefore factor it out of the integral, producing

$$\begin{aligned} I_P = {} & V_x \int_a^b \rho\,(\mathbf{P}(t))\, C(\mathbf{P}(t))\, [\tau/2 + \tau/2\,(D_x(\mathbf{P}(t)))]\, dt \\ & + V_y \int_a^b \rho\,(\mathbf{P}(t))\, C(\mathbf{P}(t))\, [\tau/2 + \tau/2\,(D_y(\mathbf{P}(t)))]\, dt \\ & + V_z \int_a^b \rho\,(\mathbf{P}(t))\, C(\mathbf{P}(t))\, [\tau/2 + \tau/2\,(D_z(\mathbf{P}(t)))]\, dt \\ & + c_1 \int_a^b \rho\,(\mathbf{P}(t))\, C(\mathbf{P}(t))\, dt. \end{aligned} \tag{3.1.3}$$

where $c_1 = \tau/2 - \tau/2\,(V_x + V_y + V_z)$.

These four integrals are now in a form that can evaluated using the Fourier projection-slice theorem. Applying the projection operator defined in equation 2.1, we obtain the following expression for output image I

$$\begin{aligned} I = {} & V_x\, \Pi_V[(\tau/2 + \tau/2\, D_x(x,y,z))\, \rho\,(x,y,z)\, C(x,y,z)] \\ & + V_y\, \Pi_V[(\tau/2 + \tau/2\, D_y(x,y,z))\, \rho\,(x,y,z)\, C(x,y,z)] \\ & + V_z\, \Pi_V[(\tau/2 + \tau/2\, D_z(x,y,z))\, \rho\,(x,y,z)\, C(x,y,z)] \\ & + c_1\, \Pi_V[\rho\,(x,y,z)\, C(x,y,z)]. \end{aligned} \tag{3.1.4}$$

The algorithm is shown in figure 4. Four copies of $\rho\,(x,y,z)\, C(x,y,z)$ are created. Each is multiplied by the appropriate depth cueing function and Fourier transformed, yielding four 3D spectra. The first three spectra represent volumes that are depth cued along one of object space coordinate axes. The fourth spectrum represents the original volume. For each view, slices are extracted from each spectra, inverse Fourier transformed, weighted by the appropriate component of the current viewing direction as shown in the figure and summed, producing an image that is depth cued for that viewing direction.

To summarize, we have replaced per-voxel shading followed by projection with projection followed by per-pixel shading. This transformation is allowed because integration is a linear operator and the current viewing direction is constant for all voxels. Both formulations produce identical results, but the latter allows a more efficient implementation using Fourier projections.

3.2. Directional shading

In the previous section, we showed that a per-voxel dot product followed by projection could be replaced with projection followed by a per-pixel dot product. There are other places where dot products appear in per-voxel shading models. Most notable is probably the $|\mathbf{N} \cdot \mathbf{L}|$ product giving the irradiance on a surface of orientation **N** produced by a parallel light source originating from direction **L**. Unfortunately, the presence of the nonlinear absolute value operator prevents evaluation of this dot product using the Fourier projection-slice theorem. In this section, we develop an alternative, physically valid, shading model in which illumination from a parallel light source is replaced by illumination from a hemispherical source. It will be shown that images based on this model can be rendered efficiently using the projection-slice theorem.

Let us begin with Lambertian reflection and parallel illumination. We define a volume of albedo $\Omega(x,y,z)$ and an orientation field $\mathbf{N}\,(x,y,z)$. The latter is typically estimated

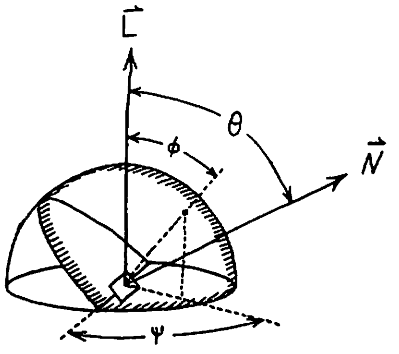


Figure 5: Hemispherical illumination of a surface. Irradiance is equal to integral of the projection of shaded portion of the hemisphere down onto the plane containing the surface.

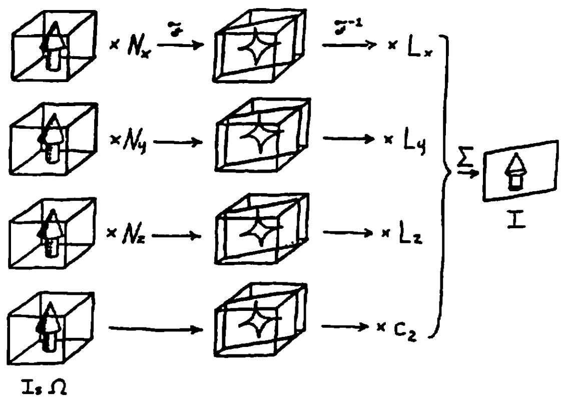


Figure 6: Algorithm for generating a directionally shaded rendering of a volume dataset. During precomputation, volume I, Ω is multiplied by orientation field N. For each view, the extracted slices are multiplied by lighting direction L.

from the scalar input data using a central difference operator [Levoy88]. Ignoring attenuation, the energy I_P arriving at point $\mathbf{P}(b)$ along viewing ray $\mathbf{P}(t)$ is given by

$$I_P = \int_a^b I_s\, \Omega(\mathbf{P}(t))\, |\mathbf{N}(\mathbf{P}(t)) \cdot \mathbf{L}|\, dt \qquad (3.2.1)$$

where I_s is the intensity of a parallel light source originating from direction L. This expression assumes illumination in both the L and -L directions. One-side illumination is obtained by replacing $|\mathbf{N} \cdot \mathbf{L}|$ with $\max(\mathbf{N} \cdot \mathbf{L}, 0)$.

Since the absolute value (or max) operator is nonlinear, we cannot factor lighting direction L out of the integral as we did with viewing direction V in section 3.1. Omitting these operators entirely would cause voxels containing backward-facing normals (relative to the light source) to produce negative light that would combine under integration with positive light produced by voxels containing forward-facing normals to yield incorrect images.

To eliminate this problem, we consider an alternative illumination method - an emissive hemisphere. The irradiance E_i on a surface having normal vector N illuminated by a hemisphere whose pole points in direction L as shown in figure 5 is equal by Nusselt's analogue (as described in [Cohen85]) to the projection of the visible portion of the hemisphere down onto the plane containing the surface, or

$$\begin{aligned} E_i &= I_s \int_{-\frac{\pi}{2}+\theta}^{\frac{\pi}{2}} \int_{-\frac{\pi}{2}}^{\frac{\pi}{2}} \cos\phi \cos^2\psi \, d\psi \, d\phi \qquad (3.2.2) \\ &= \pi/2 + \pi/2 \cos\psi \\ &= \pi/2 + \pi/2\, (\mathbf{N} \cdot \mathbf{L}). \end{aligned}$$

Assuming Lambertian reflection with albedo $\Omega(x,y,z)$, we have

$$I_P = \int_a^b I_s\, \Omega(\mathbf{P}(t))\, [\pi/2 + \pi/2\, (\mathbf{N}(\mathbf{P}(t)) \cdot \mathbf{L})]\, dt. \qquad (3.2.3)$$

Although seldom seen in the graphics literature (a notable exception is [Nishita86]), this shading model is entirely plausible - it models the appearance of diffuse objects outdoors on a cloudy day (ignoring interreflection among objects on the ground). The directional shading provided by this model is less sensitive to surface orientation than shading arising from parallel illumination, just as our perception of shape is poorer on a cloudy day than on a sunny day. Nevertheless, the model is a significant improvement over ignoring scattering effects entirely.

Continuing our derivation, we observe that equation 3.2.3 is nearly identical to equation 3.1.2. If we wish to support rotation of the volume relative to the light source yet avoid recomputing the 3D frequency domain representation for each view, the integrand must be independent of lighting direction. Looking at the equation, we see that hemisphere pole direction L is independent of ray parameter t and may be factored out of the integral. As in the case of depth cueing, this manipulation produces four integrals that can each be evaluated using the Fourier projection-slice theorem. Repeating the steps in section 3.1, we obtain the following expression for output image I

$$\begin{aligned} I &= L_x\, \Pi_V[(\pi/2 + \pi/2\, N_x(x,y,z))\, I_s\, \Omega(x,y,z)] \\ &+ L_y\, \Pi_V[(\pi/2 + \pi/2\, N_y(x,y,z))\, I_s\, \Omega(x,y,z)] \\ &+ L_z\, \Pi_V[(\pi/2 + \pi/2\, N_z(x,y,z))\, I_s\, \Omega(x,y,z)] \\ &+ c_2\, \Pi_V[I_s\, \Omega(x,y,z)] \end{aligned} \qquad (3.2.4)$$

where $N_x(x,y,z)$ is the X-component of the orientation field at location (x,y,z) similarly for N_y and N_z, and $c_2 = \pi/2 - \pi/2\,(L_x+L_y+L_z)$.

The algorithm is shown in figure 6. Four copies of $I_s\,\Omega\,(x,y,z)$ are created. Each is multiplied by the appropriate component of $\mathbf{N}\,(x,y,z)$ and Fourier transformed, yielding four 3D spectra. Because of the method used to compute $\mathbf{N}$, the first three spectra essentially represent volumes that have been gradient-enhanced in one of the object space coordinate directions. The fourth spectra represents the original volume. For each change in view or lighting direction, slices are extracted from each spectra, inverse Fourier transformed, weighted by the appropriate component of the current lighting direction as shown in the figure and summed, producing an image that is directionally shaded for that lighting direction.

As in the case of depth cueing, we have replaced per-voxel shading followed by projection with projection followed by per-pixel shading because one of the terms in a dot product - the current lighting direction - is constant for all voxels.

3.3. Depth cueing & directional shading

If depth cueing and directional shading as defined in sections 3.1 and 3.2 can each be expressed as linear combinations of Fourier projections, it should be possible to combine them, producing a composite shading model that exhibits both effects and which is expressible as a linear combination of Fourier projections.

Multiplying the integrand in equation 3.2.3 by the linear attenuation factor in equation 3.1.2, we obtain for a viewing ray

$$\begin{aligned} I_P = \int_a^b & I_s\,\Omega\,(\mathbf{P}(t))\,[\pi/2 + \pi/2\,(\mathbf{N}\,(\mathbf{P}(t))\cdot\mathbf{L})] \\ & \times\ [\tau/2 + \tau/2\,(\mathbf{D}\,(\mathbf{P}(t))\cdot\mathbf{V})]\,dt. \end{aligned} \qquad (3.3.1)$$

After we factor and apply the Fourier projection operator, we obtain for the output image

$$\begin{aligned} I &= \tau\,\pi/2\,\mathbf{V_L}\cdot\Pi_V[(1/2 + 1/2\,\mathbf{D_N}\,(x,y,z))\,I_s\,\Omega\,(x,y,z)] \\ &\quad + \tau\,\pi/2\,c_3\,\Pi_V[I_s\,\Omega\,(x,y,z)] \end{aligned} \qquad (3.3.2)$$

for $c_3 = 1/2 - 1/2\sum_i \mathbf{V}_{\mathbf{L}_i}$ where

$$\begin{aligned} \mathbf{V_L} = (&V_xL_x,\ V_xL_y,\ V_xL_z,\ V_yL_x,\ V_yL_y,\ V_yL_z, \\ &V_zL_x,\ V_zL_y,\ V_zL_z, V_x,\ V_y,\ V_z,\ L_x,\ L_y,\ L_z) \end{aligned}$$

and

$$\begin{aligned} \mathbf{D_N} = (&D_xN_x,\ D_xN_y,\ D_xN_z,\ D_yN_x,\ D_yN_y,\ D_yN_z, \\ &D_zN_x,\ D_zN_y,\ D_zN_z,\ D_x,\ D_y,\ D_z,\ N_x,\ N_y,\ N_z). \end{aligned}$$

This requires the extraction and inverse Fourier transformation of 16 slices from 16 precomputed spectra. While this algorithm is still asymptotically less expensive than conventional volume rendering, its high constant makes it slower except for very large volumes. We note that our composite shading model ignores emission $\rho\,C$ in equation 3.1.2. In the context of a directional shading model, this term can be thought of as representing ambient illumination. If included, it adds another 3 terms to equation 3.3.2 and hence another 3 slices that must be extracted.

3.4. Specular reflections and other extensions

Shiny surfaces exhibit a greater change in reflected light intensity for small variations in surface orientation than dull surfaces. A common formulation of the specular term in Phong's illumination model [Phong75] is $|\mathbf{N}\cdot\mathbf{H}|^n$ for some exponent n where $\mathbf{H}$ is the vector halfway between the directions of the light source, $\mathbf{L}$, and the viewer, $\mathbf{V}$. Using hemispherical illumination to avoid the absolute value operator, we have for a viewing ray

$$I_P = \int_a^b I_s\,\Omega\,(\mathbf{P}(t))\,[\pi/2 + \pi/2\,(\mathbf{N}\,(\mathbf{P}(t))\cdot\mathbf{H})^n]\,dt. \qquad (3.4.1)$$

After we factor and apply the Fourier projection operator, we obtain for the output image

$$\begin{aligned} I &= \mathbf{H_n}\cdot\Pi_V[(\pi/2 + \pi/2\,\mathbf{N_n}(x,y,z))\,I_s\,\Omega\,(x,y,z)] \\ &\quad + c_4\,\Pi_V[I_s\,\Omega\,(x,y,z)] \end{aligned} \qquad (3.4.2)$$

for suitable constant c_4 where $\mathbf{H_n}$ and $\mathbf{N_n}$ contain the list of factors in the polynomial $(\mathbf{N}\cdot\mathbf{H})^n$. The number of terms and hence the number of spectra is given by the multinomial theorem [Knuth73]. For $n = 10$, a typical value, we require 286 precomputed spectra, making this extension slower than conventional volume rendering even for very large volumes.

For scenes of high depth complexity, linear attenuation may provide an insufficiently steep intensity falloff to disambiguate depth relationships. Equation 3.4.1 can be adapted to provide n-degree polynomial depth cueing by replacing $\mathbf{N}$ and $\mathbf{H}$ with $\mathbf{D}$ and $\mathbf{V}$, respectively, and adjusting the constants. For example, inverse square law falloff would be given by $(\mathbf{D}\cdot\mathbf{V})^2$ and would require 10 slices from 10 precomputed spectra.

4. Implementation and results

Figures 7 through 13 each show the application of a different shading model to a 96^3 voxel cube extracted from a CT scan of a human skull mounted in a lucite head cast. The right eye orbit and a portion of the zygoma (temple) are visible at center and left, respectively, and the lucite nose is visible at lower right. To facilitate comparisons, figure 7 is a conventional volume rendering generated using the techniques described in [Levoy88].

Figures 8 through 13 were generated using the Fourier projection-slice theorem as follows. The input data was first edge enhanced and nonlinear windowed (except for figure 8 as noted below), then surrounded with a 16% margin of zeros to avoid the ghosting problem discussed in section 2. The resulting 128^3 voxel cube was weighted according to equations 3.1.4, 3.2.4, or 3.3.2, and Fourier transformed using a 3D FFT algorithm. For each view, slices of width equal to the spectra were extracted using a tricubic interpolating spline, inverse transformed using a 2D FFT algorithm, weighted appropriately for the current viewing or lighting directions, and summed to produce a 128^2 image.

Memory requirements varied from 48 megabytes for equation 3.1.4 to 256 megabytes for equation 3.3.2. Per-view rendering time varied from 6 seconds for equation 3.1.4 (about 1.5 seconds per extracted slice) to over a minute for equation 3.3.2 (due mostly to paging). Timings are for a 36 MHz Silicon Graphics 4D/320 VGX and a sequential C implementation. No attempt was made to optimize the code.

By replacing the FFT with an FHT, one can immediately halve the per-view rendering time and the memory requirements. By optimizing the filter and trimming the zero padding, another factor of two speedup and some reduction in memory usage can be expected. For greater speedups, the algorithm can be parallelized at either a fine grain (within the FFT) or a coarse grain (between slices). It is also amenable to hardware acceleration using video coprocessors or video digital signal processors (DSP). These devices frequently operate on fixed-point representations of frequency spectra, suggesting additional opportunities for reducing memory costs.

Figure 8 shows an attenuation-only projection (i.e. an X-ray) generated using a variant of equation 1.2. The lack of occlusion and shape-from-shading cues is evident. Figure 9 shows an attenuation-only projection of the input volume after edge sharpening and nonlinear windowing of the density range. Figures 10 through 13 also include sharpening and windowing. Figure 10 shows a depth-cued X-ray generated using the shading model described in section 3.1. The weighting assigned to each point in the volume falls off linearly with increasing distance from the viewer. Figure 11 shows a directionally shaded X-ray generated using the shading model described in section 3.2. Each point in the volume is weighted as if it sat on a diffusely reflecting surface illuminated by a hemisphere placed on the right side of the picture and shining leftwards. Figures 12 and 13 show two views that each include both depth cueing and directional shading using the model described in section 3.3. Although no occlusion occurs in either of these figures, the shading effects enhance our ability to perceive shape and discern spatial relationships. When displayed in motion, the visual impact is sufficiently strong that viewers often overlook the fact that these are X-rays, not opaque renderings.

5. Conclusions and future work

We have described a family of algorithms for generating realistically shaded renderings of volume data using the Fourier projection-slice theorem. While the images produced using these algorithms do not exhibit occlusion, they provide sufficient depth and shape information to make them moderately understandable in static views and very understandable in motion sequences. In either case, they produce images richer in visual cues than the attenuation-only projections with which the projection-slice theorem is usually associated.

One aspect of the Fourier approach that we have not discussed is how it impacts interactive data segmentation. The edge sharpening and nonlinear windowing operators used in figures 9 through 13 are applied before the forward 3D Fourier transform. As a consequence, changing them requires recomputing the spectra. Depending on the application, it may be undesirable to hardwire data segmentation in this way.

Several strategies are available for avoiding this difficulty. If the data segmentation operator is expressible as a linear combination of basis functions, it can be handled using the same scheme we have employed for shading models. For example, if a CT dataset is segmented into fat, muscle, and bone volumes during preprocessing, yielding three spectra, the weights and colors assigned to each material can be altered at image generation time by applying different weights to the slices extracted from each spectrum.

If the segmentation operator is expressible as a convolution, it may be possible to implement it equally or more efficiently as multiplication in the frequency domain. For example, the edge sharpening operator used in figures 9 through 13 could have been implemented as multiplication of each extracted slice by a high-frequency emphasis function. Using this approach, the amount and type of enhancement can be specified at image generation time.

Conversely, if the segmentation operator is expressible as multiplication by a function of position in the spatial domain, it may be possible to implement it as convolution in the frequency domain. For an intriguing example of this strategy, consider the linear depth cueing described in section 3.1. Although the Fourier transform of this function is infinite in extent, equally effective depth cueing can be obtained using the first half period of a suitably oriented cosine function. The Fourier transform of a cosine is two delta functions. For each view, we determine the placement of these deltas, preconvolve them with our interpolation filter (which amounts to summing two shifted copies of the filter function), and use the composite filter during slice extraction. This technique requires only one spectrum as input, making it potentially more efficient than the technique described in this paper. We are currently investigating this approach.

Looking beyond shading models, sums of Fourier projections can also be used to compute integrals over irregular spatial domains. Specifically, the integral of any function over the ray segment connecting two points in the interior of a volume can be computed as a sum of integrals over spatially disjoint intervals of the ray that taken together span the segment. Assuming well behaved functions, each integral can be computed using the projection-slice theorem. We are currently investigating an algorithm based on recursive subdivision of volumes into octants that can compute images in $O(N^2 \log^2 N)$ time. Possible applications include rendering of spatially clipped volumes for medical visualization, fast calculation of inter-element attenuation for zonal radiosity algorithms, and approximate visibility determination for geometrically defined scenes.

Acknowledgements

This research was supported by the National Science Foundation, (NSF), the National Aeronautics and Space Administration (NASA), and the sponsoring companies of the Stanford Center for Integrated Systems (CIS). The SGI 4D/320 VGX workstation was donated by Silicon Graphics, Inc. The CT scan was provided by North Carolina Memorial Hospital. Discussions with Adam Levinthal were useful in the early stages of this project, Takashi Totsuka suggested the frequency domain implementation of depth cueing, and Alice Yu provided a helpful reading of the manuscript.

References

[Bracewell86] Bracewell, R., *The Fourier Transform and Its Applications*, McGraw-Hill, 1986.

[Cohen85] Cohen, M.F. and Greenberg, D.P., "The Hemi-Cube: A Radiosity Solution for Complex Environments," *Computer Graphics (Proc. SIGGRAPH '85)*, Vol. 19, No. 3, July, 1985, pp. 31-40.

[Drebin88] Drebin, R.A., Carpenter, L., and Hanrahan, P., "Volume Rendering," *Computer Graphics (Proc. SIGGRAPH '88)*, Vol. 22, No. 4, August, 1988, pp. 65-74.

[Dudgeon84] Dudgeon, D.E. and Mersereau, R.M., *Multidimensional Digital Signal Processing*, Prentice-Hall, 1984.

[Dunne90] Dunne, S., Napel, S. and Rutt, B., "Fast Reprojection of Volume Data," *Proc. First Conference on Visualization in Biomedical Computing*, IEEE Computer Society Press, May, 1990, pp. 11-18.

[Hottel67] Hottel, H.C. and Sarofim, A.F., *Radiative Transfer*, McGraw-Hill, 1967.

[Knuth73] Knuth, D., *The Art of Computer Programming*, Addison-Wesley, 1973.

[Krueger90] Krueger, W., "Volume Rendering and Data Feature Enhancement," *Computer Graphics (Proc. San Diego Workshop on Volume Visualization)*, Vol 24, No. 5, November, 1990, pp. 21-26.

[Levoy88] Levoy, M., "Display of Surfaces from Volume Data," *IEEE Computer Graphics and Applications*, Vol. 8, No. 3, May, 1988, pp. 29-37.

[Macovski83] Macovski, A., *Medical Imaging Systems*, Prentice-Hall, 1983.

[Malzbender] Malzbender, T., "Fourier Volume Rendering." Submitted for publication.

[Max90] Max, N., Hanrahan, P. and Crawfis, R., "Area and Volume Coherence for Efficient Visualization of 3D Scalar Functions," *Computer Graphics (Proc. San Diego Workshop on Volume Visualization)*, Vol. 24, No. 5, November, 1990, pp. 27-33.

[Napel91] Napel, S., Dunne, S. and Rutt, B.K., "Fast Fourier Projection for MR Angiography," *Magnetic Resonance in Medicine*, Vol. 19, 1991, pp. 393-405.

[Nishita86] Nishita, T. and Nakamae, E., "Continuous Tone Representation of Three-Dimensional Objects Illuminated by Sky Light," *Computer Graphics (Proc. SIGGRAPH '86)*, Vol. 20, No. 4, August, 1986, pp. 125-132.

[Phong75] Bui-Tuong, Phong, "Illumination for Computer-Generated Pictures," *Communications of the ACM*. Vol. 18, No. 6, June, 1975, pp. 311-317.

[Porter84] Porter, T. and Duff, T., "Compositing Digital Images," *Computer Graphics (Proc. SIGGRAPH '84)*, Vol. 18, No. 3, July, 1984, pp. 253-259.

[Rushmeier87] Rushmeier, H.E. and Torrance, K.E., "The Zonal Method for Calculating Light Intensities in the Presence of a Participating Medium," *Computer Graphics (Proc. SIGGRAPH '87)*, Vol. 21, No. 4, July, 1987, pp. 293-302.

[Sabella88] Sabella, P., "A Rendering Algorithm for Visualizing 3D Scalar Fields," *Computer Graphics (Proc. SIGGRAPH '88)*, Vol. 22, No. 4, August 1988, pp. 51-58.

[Siegel81] Siegel, R. and Howell, J.R., *Thermal Radiation Heat Transfer*, Hemisphere Publishing, 1981.

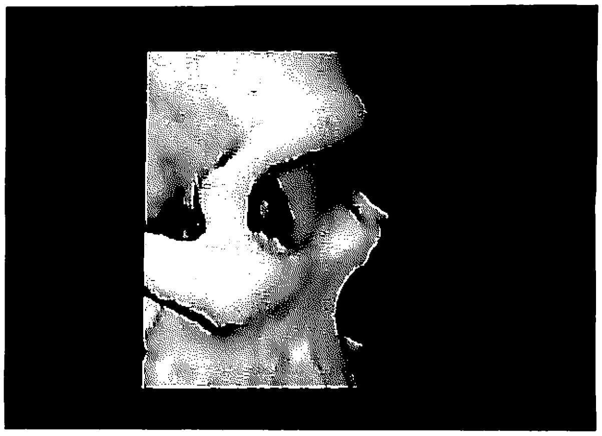


Figure 7: Conventional volume rendering of a CT scan of a human skull mounted in a lucite head cast.

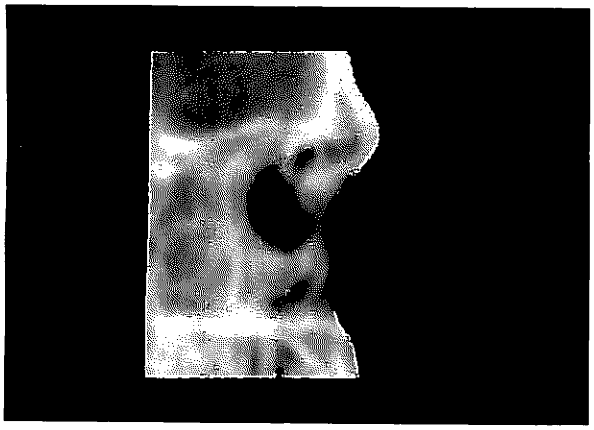


Figure 8: Attenuation-only projection (i.e. X-ray). All figures on this page were generated using the Fourier projection-slice theorem.

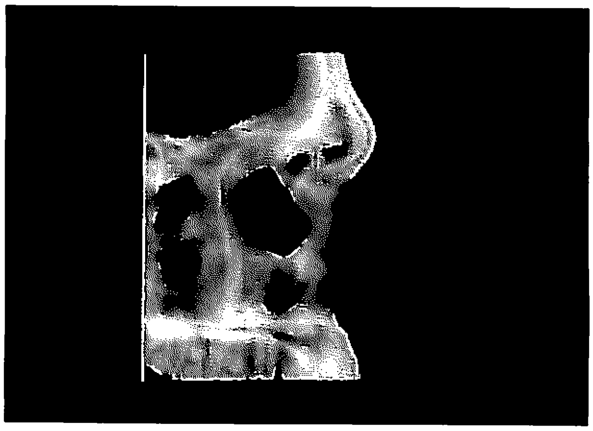


Figure 9: Attenuation-only projection after edge sharpening and nonlinear windowing. Figures 10-13 also include these enhancements.

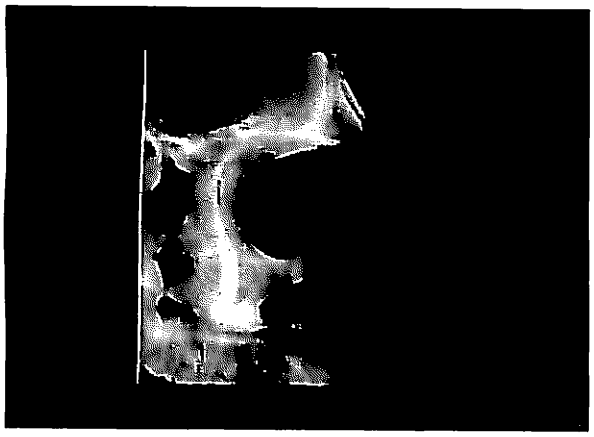


Figure 10: Depth-cued X-ray generated using the shading model described in section 3.1.

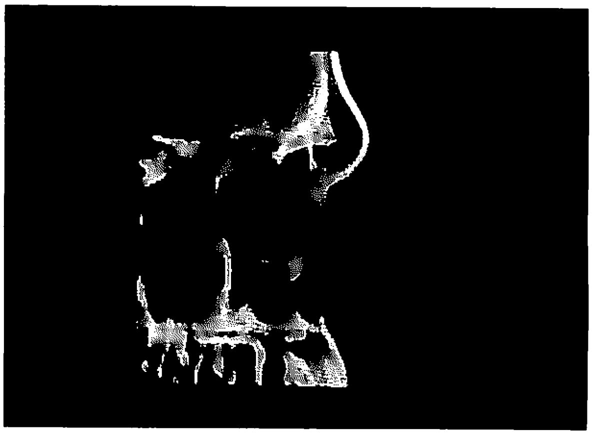


Figure 11: Directionally shaded X-ray generated using the shading model described in section 3.2.

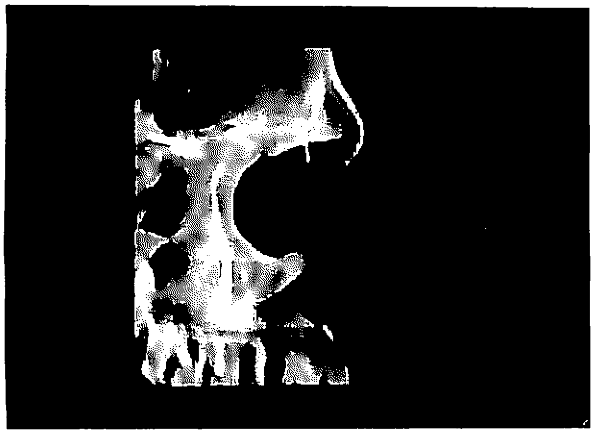


Figure 12: Depth-cued and directionally shaded X-ray generated using the shading model described in section 3.3.

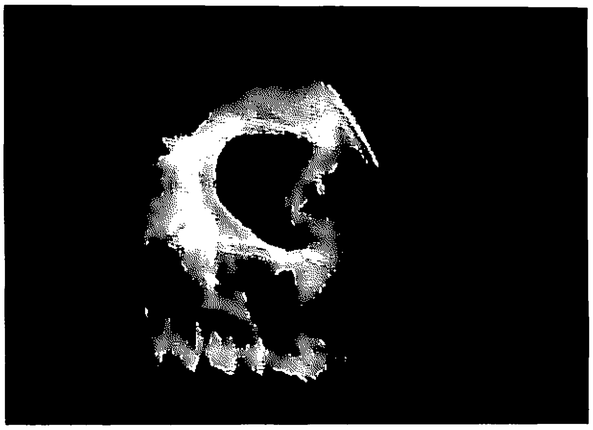


Figure 13: Rotated view of depth-cued and directionally shaded X-ray.

An Extended Cuberille Model for Identification and Display of 3D Objects From 3D Gray Value Data

Xiaoqing Qu and Wayne Davis
Department of Computing Science
University of Alberta
Edmonton Canada T6G 2H1

Abstract

This paper presents an extended cuberille model for the identification, reconstruction, and display of 3D objects from 3D gray value data. 3D edge elements are gradients detected at voxels, and the orientations of gradients are quantized to 26 directions. The edge elements are then converted to the extended cuberille model. The model has four volume primitives. Besides a cube, voxels are extended to include three other polyhedra so that voxel faces are compatible with 26 gradient orientations. The merits of the three representation schemes: space occupancy enumeration, octree, and surface representation by the extended cuberille model are briefly discussed. To identify border voxels, asymmetric Gaussian filters are applied to compute second derivative at each voxel. Conditions are defined for identifying border voxels based on the sign of the second derivative. From these conditions, there exists exactly one layer of border voxels, and subsequent surface tracking is therefore straightforward. Experimental results of 3D surface identification, tracking, and display by the extended cuberille model on test data and medical data are given. Because there are only four types of external voxel faces in the model, a surface of any object consists of only the four types of external voxel faces.

1 Introduction

This paper presents an extended cuberille model for identification, reconstruction and display of 3D objects from 3D gray value data. In 3D identification, thresholding is widely used but restricted. In a variety of applications, not many objects can be identified by simple thresholding. Identification based on 3D edge detection is a more general method. Some 3D edge operators have been proposed to detect edge elements [MR81] using gradients.

The problem of how to group detected edge elements to reconstruct an integral object has not been discussed. An integral object representation is a relatively new topic [Man88]. It implies that if an object is represented by its surface, the surface must be closed and without missing faces. If the object is represented by a collection of voxels, each voxel is a solid with a thickness so that the space occupied by the object can be measured.

Edge elements have little geometrical information because their shape and size are undefined. As a result, they cannot be used to construct an object. What is needed are modeling primitives whose shape and size are defined and whose orientations are close to their edge counterparts, i.e., a model. Currently available models are: cuberille [HL79] and a polyhedral model [LC87]. Both models use thresholding to identify objects.

In the cuberille model, an object is frequently represented as a collection of cube shaped voxels. Volume rendering algorithms can be seen in [Rey87]. An object can be represented by voxel faces. Algorithms tracking a surface of a binary image have been seen in [AFH81] and [GU89]. Because of the regularity of voxels, an object can also be organized as an octree. Octree related algorithms include tree generation algorithms [Sam80, YS83], set operation algorithms [HS79], geometric transformation algorithms [JT80, Mea82], and display algorithms [ZD91].

In the polyhedral model [LC87], a cube is made up of eight voxels in eight vertices, that are either inside or outside determined by thresholding. The algorithm marches cube by cube, creating triangle faces to model a piece of surface within each cube. A surface made of triangle faces can be displayed using a graphics package.

Some results use 3D edge detection to identify a surface. The 3D boundary following algorithm by [CR89] constructs a 3D boundary by stacking 2D boundaries, that are extracted using a graph search algorithm [Mar76].

In the following, the motivation to extend the cuberille model for 3D edge detection and surface construction is discussed.

Suppose the 3D edge operator [MR81] is applied to gray value data. It detects edges by computing the gradient at each voxel. The gradient at a voxel can be written as a vector $\nabla = (\nabla_x, \nabla_y, \nabla_z)$ with the three components indicating intensity changes along three principal axes. The magnitude of the gradient, approximated by $|\nabla| = |\nabla_x| + |\nabla_y| + |\nabla_z|$, indicates the possibility that the voxel is an edge element. The larger the magnitude, the more likely that it is an edge voxel. The direction of the gradient vector determines the edge orientation. For simplicity, the direction of a gradient is quantized to one of 26 directions (see Fig 1.)

Suppose an edge is detected at a voxel with orientation

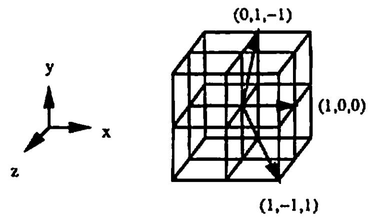


Figure 1: Three of the 26 gradient directions

(1,1,1), as shown in Fig 2(a). The cuberille model would represent the edge by three voxel faces whose average normal is (1,1,1), as shown in Fig 2(b). But it is natural to represent the edge by a face whose normal coincides with the edge orientation. For example, the triangle face in Fig 2(c) would construct a smoother surface than the three voxel faces. Of course, any other face with the same normal could also be chosen as a face primitive. So the problem is to choose a set of modeling primitives whose normals reflect the 26 gradient directions and therefore result in a smoother surface. To solve the problem the **Extended Cuberille** model is developed in the next section.

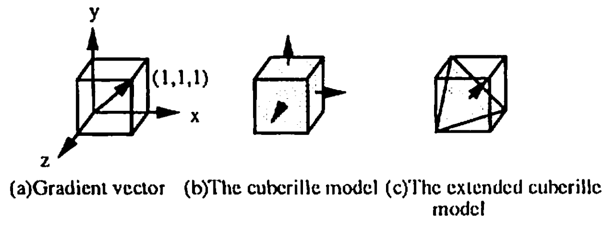


Figure 2: Different modeling primitives

2 The Extended Cuberille Model

To model voxels, the modeling volume primitives are extended to include three other polyhedra (see Fig 3) so that the voxel faces are compatible with the 26 gradient orientations.

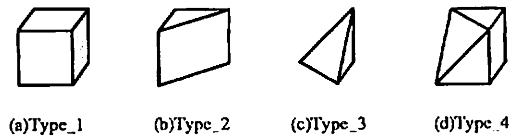


Figure 3: The set of volume primitives

As shown in Fig 3, cutting a cube voxel through diagonals of top and bottom faces gives the second polyhedron with a face orientation of (1, 0, 1). Cutting a cube voxel through three vertices defines the third and the fourth polyhedra, and gives a face oriented at (−1, 1, 1). Since the last three polyhedra are not symmetric, it is assumed that a volume primitive is invariant under 90 degree rotations about any principal axis. Hence the faces of the set of volume primitives give 26 orientations. Furthermore, to be able to represent objects hierarchically, it is also assumed that a primitive scaled by a power of two is also the same. The formal definition is given in the following.

Let V be the set of four volume primitives shown in Fig 3, i.e., V={*type_1*, *type_2*, *type_3*, *type_4*}. Let T denote the transformation of 90 degree rotations about the principal axes or power of two scalings. Thus T is an equivalent relation on V and each *type_k*, $k = 1, 2, 3, 4$, is an equivalence class by T. Let I be the 3D coordinate set, $I \subset Z^3$, Z is the set of integers, and f is a map, $f : I \to V$, then

Definition 1 *A voxel v_i is a pair of (i,f(i)), where $f(i)$ is a volume primitive, i.e., $f(i) \in V$, of minimum scale and i are the coordinates, $i \in I$.*

Arguably, a cube could be cut other ways, as shown in Fig 4, that could also result in 26 face orientations. But the set V is better than other choices for it is the smallest set that is closed under the subdivision operation.

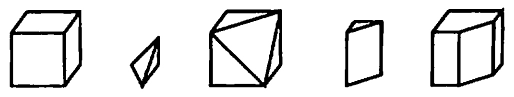


Figure 4: Another set of volume primitives

Theorem 1 *The set V is closed under the subdivision operation, and it has the least cardinality among those having the same property.*

Proof: It follows directly from subdiving the four volume primitives, as shown in Fig 5(a) to (d), that the V set is closed under subdivision. It remains to be shown that it has the least cardinality. Let U be any set of volume primitives whose type_3 and type_4 polyhedra are obtained by cutting a cube voxel, but not passing through face diagonals. Fig 5(e) shows different cuttings to get a face normal $(1, 1, 1)$. Assume the edge of a face is cut at a ratio $u/(1 - u)$, as shown in Fig 5(f), then subdivision of the cube results in a new polyhedra with edge cut ratio of $u/(0.5 - u)$. If U is closed, it must include at least two sets of polyhedra from the two ways of cutting to give the same face orientation. The same argument holds for a type_2 polyhedron. It follows $|U| > |V|$.

Since the set V is closed under subdivision, it is also possible to represent objects by octrees.

The mathematical definition of an object in the extended cuberille model is given in the following:

Definition 2 *An object S is a regularized union of voxels. $S = \cup^*_{i \in I} v_i$, where $\cup^*$ denotes the regularized union operation.*

The regularized set operations, see [Man88], defines $A \cup^* B = c(i(A \cup B))$, where $c(A \cup B)$ and $i(A \cup B)$ denote the closure and interior of $A \cup B$.

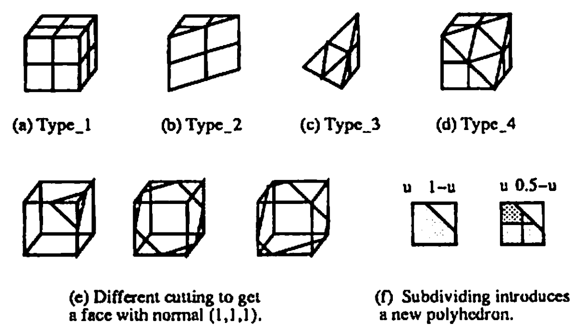

(a) Type_1
(b) Type_2
(c) Type_3
(d) Type_4
u 1-u
u 0.5-u
(e) Different cutting to get a face with normal (1,1,1).
(f) Subdividing introduces a new polyhedron.

Figure 5: The set V is closed under subdivision.

The definition gives the constructive process: because voxels can only touch in faces, edges or vertices, it implies $v_i \cap^* v_j = \phi$ for $i \neq j$. Thus $\cup^* v_i$ removes all internal voxel faces to make a solid object with one interior enclosed by a surface.

There are also several ways to represent objects in the extended cuberille model. In the next Section, the merits of three representation schemes, the enumeration scheme, octrees, and surface representation, are briefly discussed.

3 Merits of Representation Schemes

The spatial enumeration scheme in the extended cuberille model lists all voxels whose spaces are either fully or partially occupied by an object. The interior of an object is filled by type_1 voxels because they are fully occupied. All type_2 to type_4 voxels represent partially occupied space and are therefore on the border of the object. Type_1 voxels could also be on the border if the surface passes through at least one of its faces. For example, Fig 6(a) to (c) show an mathematically defined object, its space enumeration representation in the extended cuberille model, and its space occupied in the cuberille model. Although the representation gives a smoother surface, it is not as storage efficient as its counterpart in the cuberille model, for it needs two arguments - coordinates, type or gradient direction - to list a voxel.

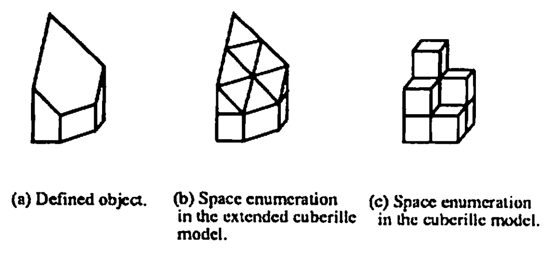

(a) Defined object.
(b) Space enumeration in the extended cuberille model.
(c) Space enumeration in the cuberille model.

Figure 6: A mathematically defined object and its representation.

An octree in the extended cuberille model may have [illegible] types of leaf nodes. In addition to the cube shaped and white leaf nodes, type_2, type_3 and type_4 leaf present partially occupied space and are all black.

An octree representing the object in the previous example is shown in Fig 7(d). Compared with the octree in the cuberille model (Fig 7(c)), the octree in the extended model is more concise because it includes leaf nodes to represent certain partially occupied space. Now consider two extreme cases shown in Fig 8. For the object in Fig 8(a), the octrees for the two models would be the same. For the object in Fig 8(b), the subdivision around the border in the cuberille model would reach the voxel level, whereas there is no subdivision in the extended model, because the root node is a leaf node. For the object with a surface slope as shown in Fig 8(c), the subdivision in the extended mode would, in the worst case, reach the voxel level. The following conclusion results:

Theorem 2 *To represent an object, the size of an octree in the extended cuberille model is at most the size of an octree in the cuberille model.*

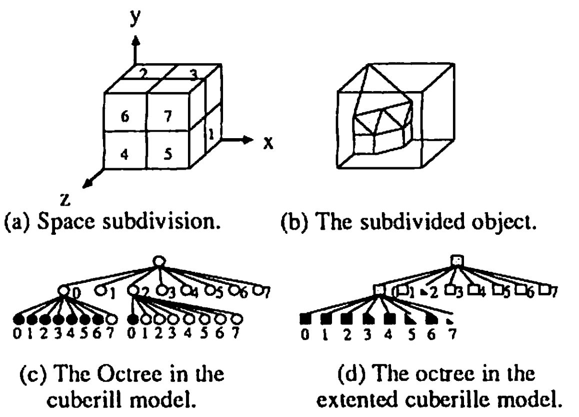

y
x
z
2
3
6
7
4
5
1
(a) Space subdivision.
(b) The subdivided object.
0 1 2 3 4 5 6 7 0 1 2 3 4 5 6 7
0 1 2 3 4 5 6 7
(c) The Octree in the cuberill model.
(d) The octree in the extented cuberille model.

Figure 7: An octree represented object in the extended cuberille model.

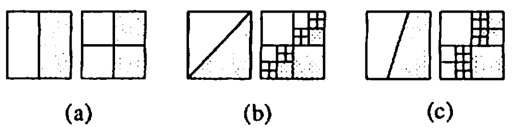


Figure 8: Comparing tree size in two models.

A surface representation lists all voxel faces on the surface. Since any object is made up of four types of voxels, and by examing Fig 3, there are only four types of voxel faces, as shown in Fig 9. It therefore follows:

Theorem 3 *The four types of external voxel faces shown in Fig 9 can close the surface of any object.*

Since there are only four types of voxel faces in the extended cuberille model, it is expected that the implementation of a surface representation is not very complicated. On the other hand, the surface of an object may not always be very smooth.

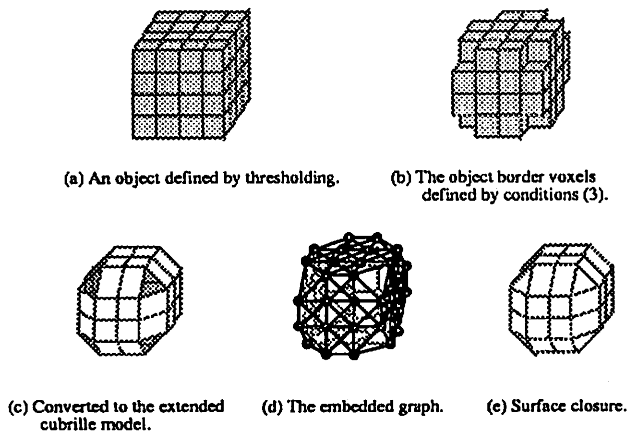

(a) An object defined by thresholding.
(b) The object border voxels defined by conditions (3).
(c) Converted to the extended cubrille model.
(d) The embedded graph.
(e) Surface closure.

Figure 13: An example to show the surface tracking steps.

```
    Surface_tracking(object)
    char *object;
    {
1         read_data(object);
2         edge_detector();
3         zero_crossing();
4         init_lists();
5         make_table();
6         Border_face_tracking();
7         Close_surface();
    }
```

Figure 14: The surface tracking algorithm

Before tracking the border voxel faces, from lines 1 to line 3, the procedure read_data() reads the data named object to an 3D unsigned character array scene. The procedure edge_detector() applies a 3D edge operator to the data array, resulting in an 3D unsigned character array grad_loc_dir of location/direction codes, and a 3D short integer array mag of gradient magnitudes. The loc_dir code of a voxel reflects the face normal that is necessary information for display. The gradient magnitude is used as a rough threshold to assist in border voxel identification. The procedure zero_crossing() computes $w(x, y, z)$ at every voxel $v(x, y, z)$, resulting in a 3D short integer array w. Implementations of these procedures are straightforward, hence no details will be given here.

In lines 4 to 5, procedures init_lists() and make_table() initialize all associated data structures, such as queue, lists, tables, etc., used by the Border_face_tracking() and Close_surface() procedures.

As shown in Fig 13(d), if border voxels and the adjacency relations between voxels can be modeled as a graph, in which nodes are border voxels and edges are adjacency relations between pairs of voxels, a standard breadth-first search can be used for tracking border voxels. The Border_face_tracking() procedure is outlined in Fig 15.

```
  int       x,y,z;
  Q_CELL    *current_voxel;

  Border_face_tracking()
  {
     unsigned char   index;

1    get_start_face;
2    span_start_face();
3    while(current_voxel=de_queue_front() {
4      x=current_voxel->x;
5      y=current_voxel->y;
6      z=current_voxel->z;
7      index=grad_loc_dir[x][y][z] & ~MARK;
8      Neighbor_face(index);
9      add_to_display_table(x,y,z,index); }
  }
```

Figure 15: The border face tracking procedure.

The underlying data structure is a queue, where each queue cell has three integers, x,y,z, for recording the coordinates of a border voxel.

In the procedure, the micro get_start_face in line 1 reads the coordinates of a start border voxel, and the procedure span_start_face() in line 2 searches its adjacent border voxels, marks them and queues them.

The while loop in line 3 starts tracking border voxels. A queue cell is obtained from the queue as the current voxel. In line 8, procedure Neighbor_face() is called to search for the current voxel's adjacent border voxels. It scans all adjacent voxels, testing if any satisfies the condition (3). Meanwhile if a border voxel is unmarked, mark it and queue it. After the procedure Neighbor_face() returns, add_to_display_table() is called to add the current voxel coordinates, x y z, and its loc_dir code, index, to a structure array to display the voxel face later. While the queue is not empty, the loop continues to the next cell in the queue. The while loop stops once the queue is empty.

The time complexity of Border_face_tracking() depends on the while loop and the procedure call Neighbor_face(). It is analyzed below. The data to start tracking are three 3D arrays, grad_loc_dir, grad_mag, and w. Bit seven of grad_loc_dir is designated for marking, hence checking and marking a voxel visited can be done in constant time. As a result, the time to execute Neighbor_face() depends on the number of adjacent voxels that a border voxel could have. Since each unmarked border voxel is placed in the queue once, the while loop is executed only once for every border voxel. Denote the number of adjacent voxels that a border voxel has as k, and the total number of the border voxels as n_B, the time complexity of Border_face_tracking() is therefore $O(kn_B)$.

For a given object, n_B is determined by the conditions (3) and is fixed. But k varies with the number of adjacent voxels that a border voxel could have. A way to define the adjacency relation between pair of voxels is by digital topology, where two voxels are adjacent each other if they are either face, edge or vertex connected. By this definition, each voxel has 26 adjacent voxels. Hence the constant k is about 26.

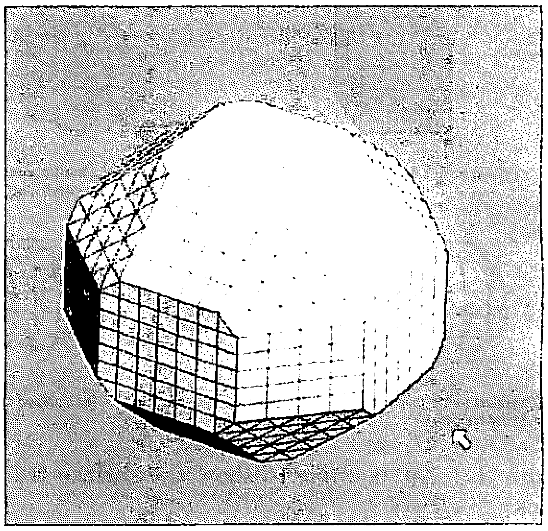
Observe from Fig 13(c), however, that in the extended cuberille model, every border voxel can be converted to a face primitive, and all the face primitives are on a surface. Intuitively, a face normal shouldn't have dramatic change from one border voxel to an adjacent one because the surface is supposed to be smooth. This suggests that the face normal of a border voxel can be used to assist in defining adjacent voxels. This results in less than 26 adjacent voxels. Furthermore, face primitives are either square or triangular face, see Fig 3, hence a face primitive has at most four edges. As a result, there are at most four ways to connect the current face to the next face. The neighbor connections are called **outways** of the current voxel. In the implementation, the Neighbor_face() procedure searches outways instead of all adjacent voxels. Whenever a border voxel of an outway is found, the search breaks and proceeds to the next outway. This reduces the constant k to about half, and speeds up border face tracking. No further details will be given. The interested reader may refer to [Qu92].

For each adjacent border voxel found for an outway, the Close_surface() procedure checks face connections between the current face primitive and the adjacent face primitive. If they are connected, do nothing. Otherwise the procedure tries to find missing voxel faces and adds the missing faces to the display table. Since procedure Close_surface() executes only for disconnected border voxel faces, it is expected to be faster than Border_face_tracking().

Back to the Surface_tracking algorithm in Fig 14, since the first three procedures work on the entire data volume, the algorithm is a volume based algorithm after all. Once voxel faces of a surface are saved in the display table, however, the time of all subsequent operations such as display, rotation and scaling, etc., operate on the border voxel faces, and therefore is an order of the number of border voxel faces of an object, i.e., $O(n_B)$.

7 Experimental Results

Fig 16 shows surfaces of a test object and a medical object obtained from real data. The test object is a sphere of volume 40 × 40 × 40 with 16 gray values. Since the test object is darker than the background, its border is a positive layer of voxels. The surface of the sphere has 509 border voxels and 797 voxel faces. It can be seen that the surfaces consist of four types of voxel faces. The images is displayed using a graphics package WINDLIB [GB87] on a Sun3/60.



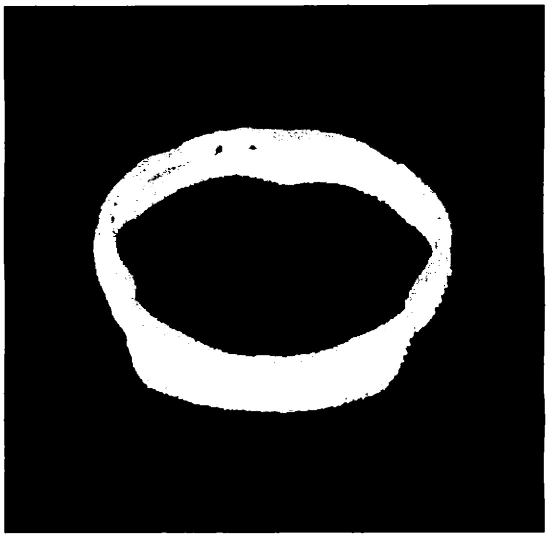


Figure 16: Surfaces of a test object and a medical object from real data.

The data size of the medical object is of 208 × 208 × 26, resulting from linearly interpolating 6 CT slices producing 5 between successive pairs. The gray values were also linerly mapped to the range from 0 to 255. The object has a brighter color than the background, therefore, its border is a negative layer of voxels. There are 17,742 border voxels detected and 26,409 voxel faces on the surface. Running **gprof** shows that the execution time for Border_face_tracking on a Sun4 is about 55.32 seconds, and checking face connections and close the surface takes 33.26 seconds.

The image is displayed on a Silicon Graphics Iris station 4D/35. To display 26,409 faces takes only seconds.

8 Conclusion

The extended cuberille model introduced in this paper provides a way to identify, reconstruct and display 3D objects based on 3D edge detection rather than thresholding.

3D edge elements are gradients, and orientations of gradients are quantized to 26 directions. The model has four volume primitives. Besides a cube, voxels are extended to include three other polyhedra so that voxel faces are compatible with 26 gradient orientations. The merits of the three representation schemes: space occupancy enumeration, octree, and surface representation by the extended cuberille model are briefly discussed.

To identify border voxels, asymmetric Gaussian filters are convolved with the gray value data to compute second derivatives of intensity changes at every voxel. Conditions for identifying border voxels based on the signs of the second derivatives are given in inequalities (3). From these conditions, there exists exactly one layer of border voxels, and subsequent surface tracking could be simply be a breadth first search.

A surface tracking algorithm using the conditions to identify border voxel is given. The surface tracking algorithm has two tasks: traverse border voxel faces and close a surface. Analysis and experimental results have shown that the the time complexity of the surface tracking algorithm depends on the border face tracking, and is an order of number of border voxels of a tracked object. Tracking outways instead of 26 adjacent voxels is further suggested to speed up border voxel tracking.

Experimental results of 3D surface identification, tracking, and display by the extended cuberille model on a test object and a medical object from real data are given. Because there are only four types of voxel faces in the model, a surface of any object consists of only four types of voxel faces.

References

[AFH81] E. Artzy, G. Frieder, and G. T. Herman. The theory, design, implementation and evaluation of a three-dimensional surface detection algorithm. *Computer Vision, Graphics, and Image Processing*, 15:1–24, 1981.

[CR89] John Danilo Cappelletti and Azriel Rosenfeld. Three-dimensional boundary following. *Computer Vision, Graphics, and Image Processing*, 48:80–92, 1989.

[GB87] M. Green and N. Bridgeman. *WINDLIB Programmer's Manual*. Department of Computing Science, University of Alberta, Edmonton, Alberta, September 1987.

[GU89] D. Gordon and J. K. Udupa. Fast surface tracking in three-dimensional binary images. *Computer Vision, Graphics, and Image Processing*, 45:196–214, 1989.

[HL79] G. T. Herman and H. K. Liu. Three-dimensional display of human organs from computed tomograms. *Computer Vision, Graphics, and Image Processing*, 9:1–21, 1979.

[HS79] G. M. Hunter and K. Steiglitz. Operations on images using quad tres. *IEEE Transactions on Pattern Analysis and Machine Intelligence*, PAMI-1(2):145–153, 1979.

[JT80] C. L. Jackins and S. L. Tanimoto. Oct-treeess and their use in representing three-dimensional objects. *Computer Vision, Graphics, and Image Processing*, 14:249–270, 1980.

[LC87] W. E. Lorensen and H. E. Cline. Marching cubes: a high resolution 3d surface construction algorithm. *ACM Computer Graphics*, 21(4):163–169, 1987.

[Man88] M. Mantyla. *An Introduction to Solid Modeling*. Computer Science Press, 1988.

[Mar76] Alberto Martelli. An application of heuristic search methods to edge and contour detection. *Communication of the ACM*, 19(2):73–83, February 1976.

[Mea82] D. J. Meagher. Geometric modeling using octree encoding. *Computer Vision, Graphics, and Image Processing*, 19:129–147, 1982.

[MH80] D. Marr and E. Hildreth. Theory of edge detection. *Proceeding of the Royol Society of London*, 207:187–217, 1980.

[MR81] D. G. Morgenthaler and A. Rosenfeld. Multidimensional edge detection by hypersurface fitting. *IEEE Transactions on Pattern Analysis and Machine Intelligence*, PAMI-3(4):187–217, July 1981.

[QD92] X. Qu and W. A. Davis. Three dimensional border identification. *Proceedings of Vision Interface'92*, page to be appear, 1992.

[Qu92] X. Qu. *Identification and Display of 3D Objects From 3D Gray Value Data*. PhD thesis, University of Alberta, to be published in 1992.

[Rey87] R. A. Reynolds. A dynamic screen technique for shaded graphics display of slice-represented objects. *Computer Vision, Graphics, and Image Processing*, 38:275–298, 1987.

[Sam80] H. Samet. Region representation: Quadtrees from binary arrays. *Computer Vision, Graphics, and Image Processing*, 13(1):88–93, 1980.

[YS83] M. Yau and S. N. Srihari. A hierarvhical data structure for multidimensional digial images. *Communication of the ACM*, 26(7):504–515, 1983.

[ZD91] J. Zhao and W. A. Davis. Fast display of octree representations of 3d objects. *Proceedings of Graphics Interface'91*, pages 160–167, 1991.

Annotating the Real World with Knowledge-Based Graphics on a See-Through Head-Mounted Display

Steven Feiner
Blair MacIntyre
Dorée Seligmann

Department of Computer Science
Columbia University
New York, New York 10027
{feiner, bm, doree}@cs.columbia.edu

Abstract

We describe an experimental, knowledge-based, virtual-world system that uses a monocular "see-through" head-mounted display to overlay graphics on the user's view of the real world. In a simple equipment maintenance domain that we have developed, the overlaid graphics include 3D representations of actual physical objects, textual annotations and callouts, and virtual metaobjects, such as arrows. A set of 3D position and orientation sensors monitor the user's head and several of the objects in the world. Our system includes a knowledge-based graphics component that determines what information to present, a display-list–based display server that represents the models to be displayed, and a set of sensor servers that track the user and selected objects. The sensor servers directly modify the object transformations and viewing specification in the display list. The knowledge-based graphics component also receives the sensor data and uses it to redesign the information being presented.

Résumé

Nous décrivons un système graphique expérimental à base de connaissances, qui enrichit la vision du monde perçue par l'utilisateur en super-imposant des graphiques générés par le système. L'utilisateur porte une lentille monoculaire semi-reflective sur laquelle s'affichent les images que notre système décide de montrer. Nous avons développé le système dans le domaine de la maintenance d'équipement. Dans ce domaine assez simple, les graphiques superimposés comportent des représentations 3D des objets réels, des annotations textuelles, et des "méta-objets" virtuels (par example, des flèches). Un ensemble de détecteurs 3D déterminent l'orientation et la position de la tête de l'utilisateur et de certains objets dans l'environnement. Notre système comprend trois composants principaux: un système de génération de graphiques à base de connaissances détermine non seulement quelles informations doivent être présentées mais aussi comment les présenter; un gestionnaire d'écran à base de liste d'objets représente les modèles des objets à afficher, et un ensemble de serveurs de détecteurs qui suivent la tête de l'utilisateur et certains objets. Les serveurs de détecteurs modifient la position des objets et les spécifications de la scène directement dans la liste des objets. Le composant graphique à base de connaissances reçoit aussi les valeurs retournées par les détecteurs et utilise cette information pour réviser l'information à présenter.

Keywords: knowledge-based graphics, virtual worlds, head-mounted displays, heads-up displays, augmented reality

1 Introduction

Virtual worlds that use 3D displays and interaction devices have the potential to make possible greatly improved explanations of complex physical tasks. These technologies promise to allow a user to view and manipulate information in ways that better exploit our ability to interact with 3D spatial information than does the use of flat 2D displays and input devices. As the richness and variety of the information that a system can present to a user increases, however, so does the difficulty of designing the presentation. Desktop publishing systems require more skill and time to master than do simple word processors; multimedia presentation editors demand yet more design expertise to use effectively. Perhaps the ultimate design demands are posed by virtual worlds, which can require the coordinated design of material that affects all sensory modalities, and that must respond continuously to the user's interactions.

Over the past years, we have been developing knowledge-based systems that address the automated design of presentations that explain how to perform simple 3D tasks. These systems generate static and animated graphics [6, 22, 14], and multimedia presentations [8] that satisfy a high-level expression of the information to be communicated. Here, we describe some first steps that we have taken toward building a testbed system for exploring the automated design of virtual worlds to explain maintenance and repair tasks. In this work, we are concentrating on the knowledge-based generation of graphics that overlay the user's view of the physical world, and which dynamically take into account information about the user, task, and changes in the surrounding world.

Ivan Sutherland, in his pioneering research on head-mounted displays, developed a binocular "see-through" system [23]. Each eye viewed a miniature vector CRT, whose synthesized

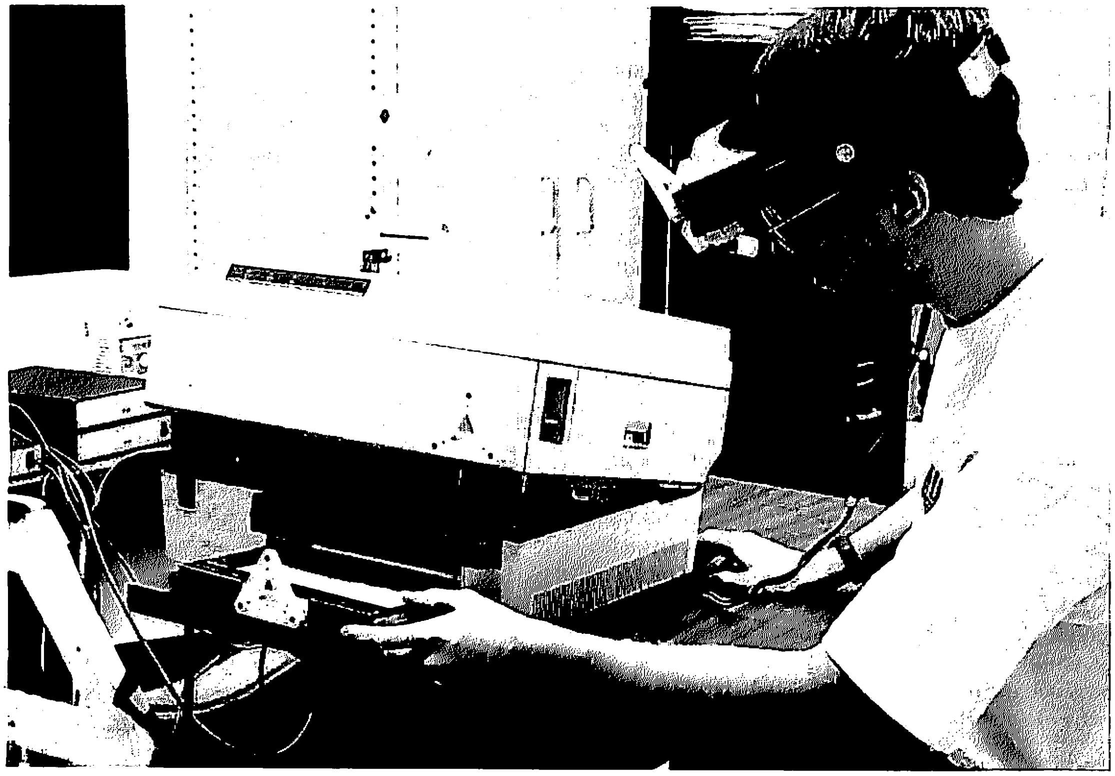


Figure 1: Prototype see-through head-mounted display in use in testbed laser printer maintenance application.

graphics were merged with the user's view of the real world by means of a beam splitter. Since then, other graphics researchers have explored the use of see-through displays, both as part of head-mounted displays (e.g., [11, 4]), and desk-mounted displays (e.g., [15, 21]). We have assembled a relatively inexpensive see-through head-mounted display, using a Reflection Technology Private Eye [20], a Logitech 3D position and orientation sensor [18], and a mirror beam splitter. This system, shown in Figure 1 in use in an experimental maintenance application described below, is a research prototype: it produces a dim, bilevel, monochrome image, and overlays graphics on a relatively narrow portion of the user's visual field, providing approximately a 22° horizontal field of view. Nevertheless, it has proven to be a useful research tool.

In the remainder of this paper, we describe the knowledge-based graphics component that designs what the user sees, a simple maintenance application with which we are experimenting, and the design and implementation of our overall system architecture.

2 Knowledge-Based Graphics Component

2.1 IBIS Overview

The knowledge-based graphics component that we use is based on IBIS (Intent-Based Illustration System) [22]. IBIS is a rule-based system that designs *illustrations*, a term that we use to refer to pictures that are designed to satisfy an input communicative intent. The communicative intent is specified by a prioritized list of *communicative goals*. Each communicative goal specifies something that the picture is to accomplish; for example, to show an object's location or shape. IBIS distinguishes between design and stylistic choice. The *design* of an illustration corresponds to its high-level structure, specifying those visual effects that must be accomplished together to satisfy the illustration's communicative goals; the *styles* used in an illustration represent the different ways each visual effect may be accomplished.

IBIS uses two kinds of rules to design an illustration: methods and evaluators. A *method* specifies how to accomplish a particular style or design; an *evaluator* determines whether a particular style or design has been accomplished. IBIS's rules allow it to examine partial solutions and to backtrack when conflicts occur.

The illustrations that IBIS generates are dynamic, rather than static: IBIS can continuously receive and handle changing constraints and goals, while at the same time realizing a particular intent. For example, IBIS normally determines a viewing specification of its own when designing an illustration. If the viewing specification is provided externally, however, then IBIS attempts to use it in generating a picture that satisfies the goals, and will incrementally redesign the picture if the viewing specification changes. This allows IBIS's user to navigate within the illustrated world by changing the viewing specification. IBIS's navigation facility is different from that used in traditional "synthetic camera" graphics in that the binding between the input intent and the output illustration is preserved. IBIS continuously examines the illustration as it is altered to attempt to satisfy the illustration's communicative goals. As the user changes the viewing specification, conflicts may occur, causing IBIS to adopt a new design or set of stylistic choices. Thus, the illustrated world itself may change as the user explores it. (A rudimentary precursor of this capability was introduced in early vector systems that made traversal of an object contingent upon the screen size of its precalculated extent, allowing an object's level-of-detail to change with its size [24].)

As a simple example of how the original version of IBIS works, consider the need to maintain object visibility. In the course of satisfying the input communicative goals, IBIS will determine that certain objects must be visible, and therefore *unoccludable*. This typically occurs if the objects participate directly in a communicative goal. Unoccludable objects must not be obscured by others in the world. IBIS can maintain the visibility of unoccludable objects by selecting an appropriate viewing specification for the illustration, by generating an inset sub-illustration, or by altering the appearance of the obscuring objects. For example, IBIS can decide not to render the obscuring objects at all, to render them as partially transparent, or to use a "cutaway" view [9].

2.2 Extending IBIS for a Head-Mounted See-Through Display

In extending IBIS to support a head-mounted see-through display, we have had to take into account a number of important differences:

- The original IBIS assumes that it generates all of what the user sees. In overlaid graphics, however, IBIS must instead enrich the user's view of the world with additional information.
- The user could modify the viewing specification after the original IBIS chose an initial viewing specification. In contrast, when designing overlaid graphics, our extended IBIS must from the beginning relinquish to the user all control of the viewing specification, since only the user can determine where they look.
- The world was assumed to change only after an illustration was completed in the original IBIS. This was easy to enforce since IBIS maintained control over what was visible, and based its illustrations on a world model that was frozen throughout the illustration's life. In contrast, our extended IBIS must take into account ongoing changes in the world as it generates illustrations.
- The original IBIS was responsible for achieving all communicative goals itself. Because the extended IBIS controls only what it generates, the user becomes an active participant in achieving the communicative goals. For example, if a goal is to specify an object's position, and the object's projection does not lie within the window, IBIS must indicate to the user where they should look.

Based on these differences, we have developed a preliminary set of new IBIS rules to handle the input communicative goals. At its core is a set of low-level style rules, each of which may be invoked for any object. These include:

- Make an object *visible*. If the object's projection falls within the overlay window and is not occluded by other objects, nothing need be done: the object is not rendered. (In contrast, in the original IBIS, the object would have to be rendered.) If the object's projection is within the window, but is occluded wholly or in part by other objects, the object is marked for rendering to allow it to be seen through its occluders; the object's rendering style will be determined by other rules. (Occlusion tests are performed with a fast image-precision algorithm described in [9].) If the object does not project to the window, the goal fails.[1]
- *Highlight* an object. If the object is within the window, the object is marked for rendering and tagged with a highlighted style (e.g., solid lines); otherwise, the goal fails.

Representative higher-level design rules that invoke these are:

- *Show* an object. If the object is *visible*, nothing need be done; otherwise, the *find* goal is activated.
- Show the *location* of an object. If the object is *visible*, it is *highlighted*; otherwise, the *find* goal is activated.
- *Find* an object. A callout (currently, a "canned" text string label) is created for the object, fixed in 2D relative to the overlay window. A dotted rubberband leader line connects the callout to the object. (If the object does not project to the overlay window, then the dotted line extends to the edge of the window in the direction of the object, and can be followed by the user to find the object; the callout and its end of the leader line is always in view.)
- Show a *change* to be accomplished in an object's state. This is achieved by generating a "ghost image" of the object in the desired state.
- Show an *action* performed on an object. This is achieved by adding a *metaobject*, such as an arrow, to depict the action (e.g., pushing or opening).
- *Identify* an object. If the object is *visible*, a 3D callout (textual label) is generated near the object; otherwise the *find* goal is activated.

IBIS is initially provided with representations of the physical objects in the environment. It uses these representations both to generate the overlaid graphics and to evaluate it, employing

[1]All goals succeed except where failure is indicated explicitly.

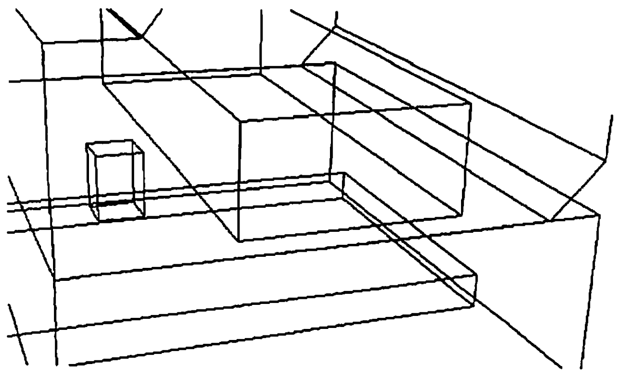


Figure 2: Overlay graphics of simple world model.

the approaches described in [22, 9].

3 Testbed Application: End-User Laser Printer Maintenance

To test our ideas, we have been experimenting with a simple end-user maintenance application for a laser printer. This entails relatively straightforward operations, such as refilling the paper tray and and replacing the toner cartridge.

Figure 1 shows the system in use. The large white triangles toward the top and bottom of the figure are Logitech sensor transmitters, which have three small ultrasonic sources near their vertices. Small triangles mounted on the display and on the laser printer each contain three microphones. The top transmitter is for the head; the bottom transmitter is for the sensors that track the printer's paper tray and lid.

Figure 2 shows part of our simple laser printer world as displayed in our head-mounted display's overlay bitmap without any design done by IBIS. In contrast, Figure 3 shows an example of an overlay illustration designed by IBIS, using the same viewing specification, to fulfill the *location* and *find* goals for the paper tray and the *show* goal for the toner cartridge. To achieve the *location* goal, the paper tray is highlighted (drawn solid), since the *visible* goal succeeded. To achieve the *find* goal, a 2D label is generated with a dotted leader line that terminates on the tray. This illustration also *shows* the toner cartridge—since it is occluded by the obscuring printer cover, it is drawn with dashed lines to make it visible. Figure 4 *shows* the paper tray, shows the desired *action* of pulling out the paper tray by means of a *metaobject* arrow, and shows the *change* in the paper tray's location that would result as a dotted "ghost" image.

4 System Architecture

The original version of IBIS synchronously designs and then renders each illustration, and then incrementally redesigns the picture in response to user interaction. While the resulting delay is barely tolerable in fully-synthesized animation displayed on a conventional CRT, it is unusable if synthesized graphics must not only change in response to head motion, but must be registered with objects in the real world. In addition, we also need to process data from a number of motion trackers. Based on our own experience [7] and that of other researchers [1, 12, 17], it was clear to us that it would be necessary to distribute processing over multiple processes and processors.

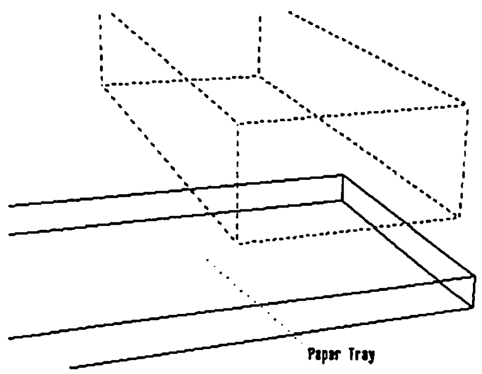


Figure 3: Overlay graphics designed by IBIS to *find* and show *location* of paper tray and *show* toner cartridge.

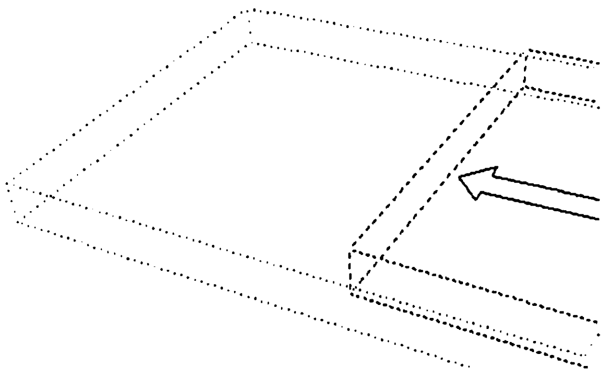


Figure 4: Overlay graphics designed by IBIS to show *action* of pulling out paper tray and resulting *change* in tray's state.

The Private Eye, in its 720×280 resolution mode, presents an interface that appears as a bare single-bit-deep framebuffer. We created a small 3D structured display-list-based *display server*. The server supports line and text primitives, linestyle and font attributes, and a set of structure management operators that allow hierarchical objects to be created, edited, and deleted. Positions in the display server's move and draw commands may be either 2D device coordinates or 3D world coordinates, and may be intermixed in a single primitive, allowing the creation of lines that are anchored to the screen on one end, and to a point in the 3D world on the other. As shown in Figure 5, when the system is initialized, IBIS creates the initial display list by sending a set of object models to the display server.

Each tracker is handled by a low-level *tracker process*. These processes in turn interact with a set of object servers and a head server. Each *object server* is associated with an object in the real world that is monitored with a tracker. At initialization, IBIS provides each object server with the identifier of the display server structure containing the object's vector representation. It also tells the server the position and orientation of the object's tracker relative to the object's coordinate

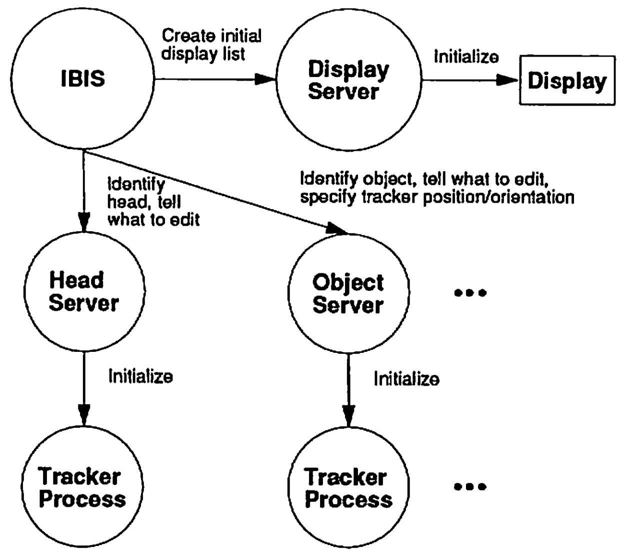

IBIS
Create initial display list
Display Server
Initialize
Display
Identify head, tell what to edit
Identify object, tell what to edit, specify tracker position/orientation
Head Server
Object Server
Initialize
Initialize
Tracker Process
Tracker Process

Figure 5: System architecture: Initialization

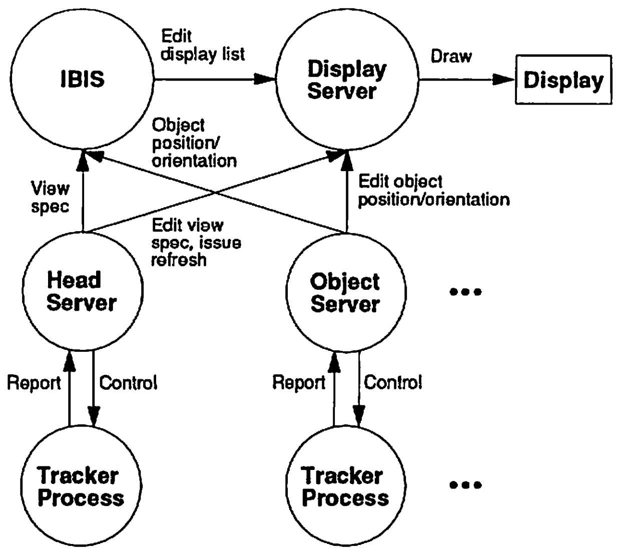


Figure 6: System architecture: Steady state

system. Similarly, IBIS provides the *head server* with the identifier associated with the scene in the display server. (The head and object servers actually use the same executable, so each must also be told whether it is tracking the head or an object.)

Figure 6 shows the system's operation after initialization. The head and object servers are responsible for maintaining the integrity of the object and head motion information that they represent. They edit the display list stored in the display server directly. Each object server regularly edits the display list position and orientation information associated with its object. The head server updates the display list viewing specification for the scene. Both head and object servers also report their information to IBIS. This avoids the delay that would result if IBIS were to serve as a go-between, making possible relatively smooth visual response to head and object motion, while assuring that IBIS always has the latest information on which to base its illustration design.

IBIS determines the presence and appearance of the objects in the display list—all information for which it has not explicitly relinquished control to the head and object servers. This includes the specification of *metaobjects*, such as arrows and text. (IBIS can also dynamically reassign tasks to the head and object servers, a facility that we have not yet used.) In work on the automated design of multimedia presentations, IBIS is presented with a set of communicative goals to satisfy, which are output by other components that determine what to say and which media should be used to say it [8]. In the research reported on here, we have instead used a much simpler content planner to generate the set of communicative goals that IBIS receives.

IBIS first designs an illustration that satisfies the initial set of goals set by its content planner, obeying the constraints imposed by the head and object trackers. Then it loops, using its evaluators to determine whether the current illustration design is satisfactory in the face of changes in the goal set, the user, and the world. Whenever the illustration is determined to be unsatisfactory, IBIS's rules result in modifications that are communicated to the display server. For example, if the 3D label of an object that IBIS determines must be identified no longer lies within the overlaid graphics window, then IBIS will generate a new 2D screen label and 3D leader line that points to the object.

Because IBIS has relinquished to the head and object servers the straightforward matters of display-list traversal and updates of the viewing-specification and monitored object transformations, user interaction with the current illustration stored in the server is fully interactive, and occurs while incremental redesign takes place on a separate processor. We currently achieve about 15 frames per second for a model containing 70 displayed vectors, but with some jerkiness due to network contention.

The head server is responsible for instructing the display server to render the image. Since polling the head's motion tracker requires much less time than rendering the image, we need to avoid building up a backlog for the display server. We accomplish this by having the head server send its commands for the current frame to the display server and then wait until it receives the display server's acknowledgment that the *previous* frame was rendered. This allows rendering and polling to proceed in parallel. All of the actions requested of the display server (e.g., modifying the contents of an object) are processed atomically to ensure that the scene always appears in a valid, drawable state.

5 Implementation

The components of the system run on several different machines under different flavors of UNIX, and communicate through sockets. IBIS is implemented in C++ and the CLIPS production system language [5], and runs under HPUX on an HP 9000 380 TurboSRX graphics workstation, which provides a fast hardware z-buffer–based graphics accelerator that IBIS uses in its illustration design process. The display server is written in C and runs under Mach on an Intel 486-based PC "clone," which supports the Private Eye display entirely in software. (A significant portion of the display server's time is spent implementing double buffering, copying the graphics frame buffer to the Private Eye's frame buffer and clearing it for the next frame.) The head/object server and lower level tracker processes are written in C and C++. Since the tracker hardware requires only an RS232 interface, and the servers can impose a large load on the machine on which they execute, we run them on other workstations.

Our current 6-DOF tracker devices include three Logitech ultrasonic sensors [18] and one Polhemus magnetic sensor [16]. Our software allows both kinds of 6-DOF sensors to be used interchangeably, for example to trade off an ultrasonic sensor's freedom from magnetic interference against a magnetic sensor's ability to work without a direct line of sight to its source.

One important point, that others have noted before [2], is that experimental interfaces that are coupled tightly to the user often require a fair amount of calibration. Our head-mounted display is no exception. Since the generated image must be registered with that of the real world, each user must perform three kinds of calibration:

- *focus*. The Private Eye is focusable from less than a foot to infinity, with each user requiring separate focus adjustments based on their eyesight. After putting on the display, the user must position a slider until a calibration image is comfortably in focus (but see below).
- *visible area*. Due to the physical relationship between the Private Eye, the beam splitter, and the current focus setting, a small portion of the display may not to be visible to a viewer. Therefore, we request that the user determine the viewable area by adjusting the size and position of a visible rectangle until it is as large as possible. This establishes a "safe-title" area that is communicated to IBIS, in which IBIS can assume that anything that is drawn will be visible.
- *viewing specification*. To register the image with the world, the user must first physically adjust the display on their head. Next, they must register a virtual object with its corresponding physical object, which in turn must be viewed in a known position relative to the user. The software takes into account the difference in position and orientation of the user's eye and the measured position and orientation of the sensor.

Registration is a serious issue: Our current motion trackers have neither the spatial nor angular resolution needed to register graphics precisely with the surrounding world. However, we have yet to try mapping sensor inaccuracies to correct for nonlinearities [21], and believe that we can also improve accuracy with a better calibration strategy. (Registration is currently off by about an inch in world coordinates in Figures 2–4.)

We have also found focus to be a particular problem in our current application. We originally built our head-mounted display for a *hybrid user interface*, which embeds the user's view of a "notebook" flat-panel display inside a partial spherical information surround presented on the head-mounted display [10]. The flat-panel display is relatively small and tangent to the sphere, which is centered about the user's head. Assuming that the user's head is stationary, it is easy to adjust the Private Eye so that the material that it displays is focused at the same distance from the user as the flat-panel display. In comparison, "heads up" flight displays are focused at infinity [19], since this is effectively where out-of-the-cockpit targets are located.

In contrast to both, our application requires that graphics be overlaid on nearby objects that necessitate constant changes in visual accommodation to focus in sequence. Since the Private Eye must be focused manually, the need for readjustment as the viewing distance to the object of interest changes is irritating.[2] There is an interesting benefit, however, to the precise focus control provided by the Private Eye. Even though we are currently using a monocular display, when the display is adjusted so that a small synthesized object is in focus at a particular distance, the illusion of it being at the selected position is quite compelling. (Note that one of the difficulties that many users experience in viewing fixed focus stereo displays is developing independence between their ocular convergence and focus accommodation.)

6 Conclusions and Future Work

The work that we have reported on here represents our first steps in designing a testbed for the knowledge-based generation of maintenance and repair instructions using a head-mounted, see-through display. We have developed a preliminary set of rules that allow us to augment the user's view of the world with additional information that supports the performance of simple tasks such as finding designated objects and carrying out simple actions on them. Our software architecture makes a clean distinction between design and rendering to help prevent design decisions from interfering with interactive rendering.

Our experience with the system has suggested many research directions that need to be explored. For example, one important problem is the development of a formal model of how a user's performance will be affected by different decisions made in designing 3D illustrations, taking into account the purpose for which the illustration is generated (specified by our communicative goals), as in the 2D design work of Casner [3].

[2]We have considered automating the process by using a servomotor to adjust the focus to that of a selected object. Note, however, that focus in the entire overlay would be affected uniformly.

Support for visible-line and visible-surface determination is another issue. IBIS currently bases its illustration design in part on whether selected objects are occluded in the current viewing specification, and computes these relationships itself. Our display server, however, does not support visible-line determination. We are particularly interested in incorporating into the display server what Kamada and Kawai [13] refer to as "picturing functions," which determine how a projected line fragment should be rendered, based on the set of surfaces that obscure it. For example, while IBIS currently sets the graphical attributes of an entire object based in part on its visibility, we would like more precise control to be accomplished by the display server, based on the visibility relationships of each line fragment. IBIS would then be responsible for determining the high-level policies used by the display server (e.g., making obscured parts dashed). One challenge is to do this while still maintaining real-time performance. We believe this could be possible on our current hardware if IBIS were allowed to select a subset of objects against which the display server would perform visibility tests. Another intriguing research direction is to explore how to design support facilities that would allow IBIS to specify a rich set of additional high-level policies to be enforced by the display server.

Acknowledgments

Research on this project is supported in part by the Office of Naval Research under Contract N00014-91-J-1872, the Center for Telecommunications Research under NSF Grant ECD-88-11111, NSF Grant CDA-9022123, and an equipment grant from the Hewlett-Packard Company. Work on the original version of IBIS was supported in part by the Defense Advanced Research Projects Agency under Contract N00039-84-C-0165. Our work owes much to the portable computing infrastructure software developed for the Columbia Student Electronic Notebook project directed by Daniel Duchamp, Steven Feiner, and Gerald Maguire. We thank Ari Shamash and Sushil Da Silva for their work on the Logitech 6-DOF tracker process and bitmap scan-conversion package, Brad Paley for sharing his tracker support code, Cliff Beshers for his help in porting his distributed DataGlove server, and Jim Barnes of Logitech for his assistance. Michael Elhadad et Colette Johnen nous ont aidés à traduire le résumé.

References

[1] Blanchard, C., Burgess, S., Harvill, Y., Lanier, J. Lasko, A., Oberman, M., and Teitel, M. Reality Built for Two: A Virtual Reality Tool. In *Proc. 1990 Symp. on Interactive 3D Graphics (Computer Graphics, 24:2, March 1990)*, pages 35–36. Snowbird, UT, March 25–28, 1990.

[2] Brooks Jr., F. Grasping Reality Through Illusion—Interactive Graphics Serving Science. In *Proc. CHI '88*, pages 1–10. Washington, DC, May 15–19, 1988.

[3] Casner, S. A Task-Analytic Approach to the Automated Design of Graphic Presentations. *ACM Transactions on Graphics* 10(2):111–151, April, 1991.

[4] Chung, J., Harris, M., Brooks, F., Fuchs, H., Kelley, M., Hughes, J., Ouh-young, M., Cheung, C., Holloway, R., and Pique, M. Exploring Virtual Worlds with Head-Mounted Displays. In *Proc. SPIE Non-Holographic True 3-Dimensional Display Technologies, Vol. 1083*. Los Angeles, January 15–20, 1989.

[5] Culbert, C. *CLIPS Reference Manual* NASA/Johnson Space Center, TX, 1988.

[6] Feiner, S. APEX: An Experiment in the Automated Creation of Pictorial Explanations. *IEEE Computer Graphics and Applications* 5(11):29–38, November, 1985.

[7] Feiner, S. and Beshers, C. Worlds within Worlds: Metaphors for Exploring *n*-Dimensional Virtual Worlds. In *Proc. UIST '90 (ACM Symp. on User Interface Software)*, pages 76–83. Snowbird, UT, October 3–5, 1990.

[8] Feiner, S. and McKeown, K. Automating the Generation of Coordinated Multimedia Explanations. *IEEE Computer* 24(10):33–41, October, 1991.

[9] Feiner, S. and Seligmann, D. Dynamic 3D Illustrations with Visibility Constraints. In Patrikalakis, N. (editor), *Scientific Visualization of Physical Phenomena (Proc. Computer Graphics International '91, Cambridge, MA, June 26–28, 1991)*, pages 525–543. Springer-Verlag, Tokyo, 1991.

[10] Feiner, S. and Shamash, A. Hybrid User Interfaces: Breeding Virtually Bigger Interfaces for Physically Smaller Computers. In *Proc. UIST '91 (ACM Symp. on User Interface Software and Technology)*, pages 9–17. Hilton Head, SC, November 11–13, 1991.

[11] Fisher, S., McGreevy, M., Humphries, J., and Robinett, W. Virtual Environment Display System. In *Proc. 1986 Workshop on Interactive 3D Graphics*, pages 77–87. Chapel Hill, NC, October 23–24, 1986.

[12] Green, M. and Shaw, C. The DataPaper: Living in the Virtual World. In *Proc. Graphics Interface '90*, pages 123–130. Halifax, Nova Scotia, May 14–18, 1990.

[13] Kamada, T. and Kawai, S.
An Enhanced Treatment of Hidden Lines.
ACM Transactions on Graphics 6(4):308–323, October, 1987.

[14] Karp, P. and Feiner, S.
Issues in the Automated Generation of Animated Presentations.
In *Proc. Graphics Interface '90*, pages 39–48. Halifax, Canada, May 14–18, 1990.

[15] Knowlton, K.
Computer Displays Optically Superimposed on Input Devices.
The Bell System Technical Journal 56(3), March, 1977.

[16] Kuipers, J.B.
SPASYN—A New Transducing Technique for Visually Coupled Control Systems.
In *Proc. Symp. on Visually Coupled Systems: Development and Application.* Brooks AFB, TX, September, 1973.
AMRL/WPAFB Report No. AMD TR-73-1.

[17] Lewis, B., Koved, L., and Ling, D.
Dialogue Structures for Virtual Worlds.
In *Proc. CHI '91*, pages 131–136. ACM Press, New Orleans, LA, April 27–May 2, 1991.

[18] Logitech, Inc.
Logitech 2D/6D Mouse Technical Reference Manual (Preliminary).
Fremont, CA, 1991.

[19] Norman, J. and Ehrlich, S.
Visual Accommodation and Virtual Image Displays: Target Detection and Recognition.
Human Factors 28(2):135–151, 1986.

[20] Reflection Technology.
Private Eye Product Literature.
Waltham, MA, 1990.

[21] Schmandt, C.
Spatial Input/Display Correspondence in a Stereoscopic Computer Graphic Work Station.
Computer Graphics (Proc. SIGGRAPH '83) 17(3):253–261, July, 1983.

[22] Seligmann, D. and Feiner, S.
Automated Generation of Intent-Based 3D Illustrations.
In *Proc. ACM SIGGRAPH '91 (Computer Graphics, 25:4, July 1991)*, pages 123–132. Las Vegas, NV, July 28–August 2, 1991.

[23] Sutherland, I.
A Head-Mounted Three Dimensional Display.
In *Proc. FJCC 1968*, pages 757–764. Thompson Books, Washington, DC, 1968.

[24] van Dam, A., Stabler, G., and Harrington, R.
Intelligent Satellites for Interactive Graphics.
Proc. IEEE 64(4):483–492, April, 1974.

aliasing we must detect situations during rendering where we go above this sampling step size.

Consider the geometry of shading:

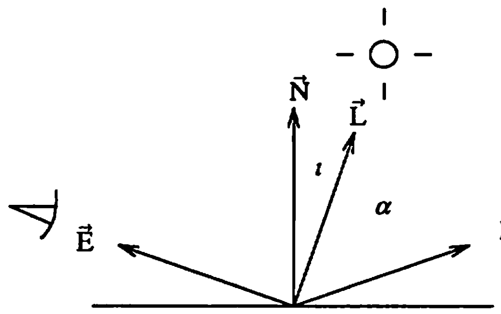


Figure 1

Here, $\vec{N}$, $\vec{L}$, $\vec{E}$ and $\vec{R}$ are unit vectors in the direction of the surface normal, light source, eye and reflected eye directions respectively. Changing any of $\vec{N}$, $\vec{L}$ or $\vec{E}$ will change α, the α of Phong's highlight function. If any of these change within a pixel when rendering, specular aliasing may occur (Williams, 1983). Luckily, unless the light source or eye are very close to the surface, their effect is minimal. The surface normal, however, has a more pronounced effect. It can change significantly within a pixel, thus noticeably affecting α. If the surface is shiny (a high value of n), the change in α ($\Delta\alpha$) within the pixel can exceed $\Delta\alpha_n$ (which is smaller for larger values of n) and aliasing results. Consequently, to detect specular aliasing, we must, within each pixel, determine how much the surface normal $\vec{N}$ changes and relate it to the maximum allowed by n. (Note: a change of $\vec{N}$ by Δ radians changes α by 2Δ radians).

The most straightforward way to compute how much the surface normal changes is to compute the normal at the corners of the pixel and find the pair which diverge the most (by computing the six dot products of the various pairs of normals and finding the minimum). The smallest of these six dot products, $dot_{\Delta\vec{N}}$, is compared to the smallest dot product allowed for the current value of n, $dot_{\Delta\alpha_n}$ (via a lookup on n into a pre-computed table). If it is greater than the value in the table, then no aliasing can occur. Otherwise, we have detected the occurrence of specular aliasing. We thus have derived a simple, analytic algorithm to detect specular aliasing on surfaces using the Phong highlight function. The only prerequisite is a good indicator of $\Delta\alpha$. Instead of using the normals at the corners of the pixel, another possibility would be to use the curvature of the surface and the size of intersection to compute $\Delta\alpha$.

The above approach of finding the how much the surface normal changes within a pixel has to be modified slightly when bump mapping is used. In this case, one can compute the change in normal when the bump map normal is generated and use this value instead.

The algorithm described above is conservative. Most of the energy in the power spectrum of $\cos^n(\alpha)$ is in the lower frequencies with very little near the high end. Consider the following figure:

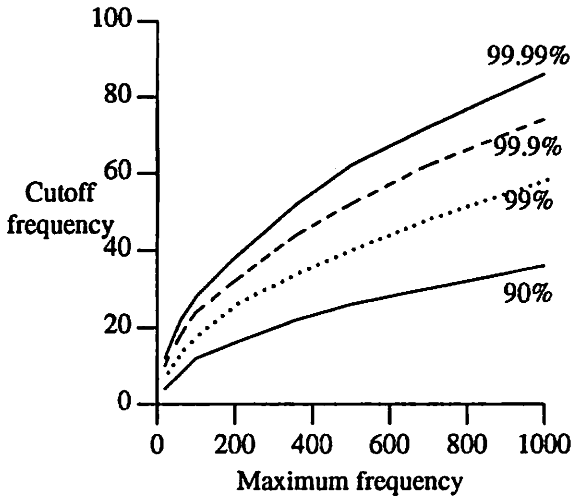


Figure 2

It is a frequency-frequency plot for several power levels. The abscissa indicates the maximum frequency in the power spectrum of $\cos^n(\alpha)$ (recall that the maximum frequency is n radians) while the ordinate represents the cutoff frequency below which a given percentage of the power of $\cos^n(\alpha)$ resides. For example, if n is 500, the maximum frequency in the power spectrum is 500 radians while 99.9 percent of the power is below 52 radians. Our table of minimum dot products could be computed more aggressively so that we don't have to perform specular anti-aliasing as frequently.

Once we have detected that specular aliasing is occurring we must be able to suggest a sampling rate that eliminates it. To do this we take the arc-cosine of the normal-pair dot product, $dot_{\Delta\vec{N}}$, (for efficiency we could perform a table lookup for the arc-cosine) to get $\Delta\alpha_{\Delta\vec{N}}$. We compare this with $\Delta\alpha_n$ and use the resultant ratio to indicate the oversampling rate.

Eliminating Specular Aliasing

We have just derived an algorithm for the detection of specular aliasing. Now we have to decide what to do about it. Crow's standard solution for removing this aliasing can be used but it is unsatisfactory in that multiple shading computations per pixel per polygon are required; these are expensive. We will now derive an alternate solution that requires only one shading computation per pixel, a much more frugal approach.

The purpose of anti-aliasing is to remove the high frequencies in a signal that cannot be represented by the current sampling rate. If we do not want to change this sampling rate we must somehow change the signal so that the unrepresentable frequencies are absent. One way of accomplishing this low pass filter is described by Norton, Rockwood and Skolmoski in the work they did on texture mapping (Norton, 1982). Their original signal was constructed with a series of sine waves, each having a different frequency and phase angle. Their method of anti-aliasing was to clamp out the sine waves that were too high to be represented and only use the low frequency sine wave components. This approach inspired the specular anti-aliasing algorithm described below.

A different method to perform specular anti-aliasing involves clamping n to values that will not introduce aliasing. By replacing n by a smaller value, we guarantee that the new highlight function has no offending high frequencies. The new value of n, n', depends on the sampling rate $\Delta\alpha_{\tilde{N}}$. What we are in fact doing is replacing the user specified highlight function with a duller one, one guaranteed not to alias. We only do this, however, in problem pixels so that in regions where no aliasing is occurring we use the original function.

The simple replacement of n' for n needs a little enhancement before it becomes the complete anti-aliasing algorithm. Replacing $\cos^{n'}(\alpha)$ for $\cos^{n}(\alpha)$ removes the high frequencies from the original signal but in the process it also boosts the lower frequencies, making the surface appear brighter than before. Some sort of normalization is in order. This normalization is encorporated in the algorithm if we multiply the new highlight function, $\cos^{n'}(\alpha)$, by the ratio of Maximum[n] to Maximum[n']. Maximum[] is an array that stores the value of the first (and largest) component in the signal $\cos^{n}(\alpha)$ for various values of n. By performing the above normalization we try to make sure that the overall level of the clamped signal is the same as the original signal. The effects of normalization is illustrated in Figure 6-3. It plots the ratio of the cosine components of the clamped signal to those in the original signal, $\cos^{160}(\alpha)$, for various values of n', starting at 20 and going to 140 in step sizes of 20. We see that in the low frequencies the original and clamped components are identical but as we go higher up in the frequency spectrum the cosine components in the clamped function quickly fall off.

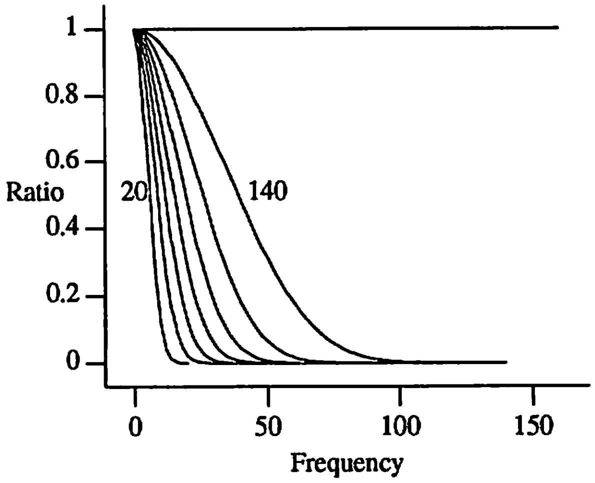


Figure 3

The resulting specular anti-aliasing algorithm is summarized in the code fragment below:

```
if($dot_{\Delta\tilde{N}}$ >= MinDot[$n$]) {
    /* no aliasing */
    $n' = n$;
    $K_{specular}' = K_{specular}$;
} else {
    $\Delta\alpha_{\tilde{N}}$= 2*ArcCosine($dot_{\Delta\tilde{N}}$);
    sampleFrequency= $\pi/\Delta\alpha_{\tilde{N}}$;
    $n'$= Max$n$Allowed[sampleFrequency];
    $K_{specular}' = K_{specular}$*Maximum[$n$]/Maximum[$n'$];
}
```

$K_{specular}$ indicates the fraction of the light that the specular component contributes. The array Max$n$Allowed[] is required only if we are performing aggressive specular anti-aliasing. Otherwise, n' is assigned the value of sampleFrequency.

Results

The above algorithms for detecting and eliminating specular aliasing were implemented in a z-buffer rendering system. The z-buffer visible surface algorithm was chosen because it would guarantee that any anti-aliasing observed would have to come from the clamping algorithm. Adding the clamping algorithm to the z-buffer renderer was straightforward. The biggest implementation hurdle encountered was changing the tiler to have it compute $dot_{\Delta\tilde{N}}$. The three arrays, MinDot[], Maximum[] and Max*n*Allowed[], were pre-computed for efficiency. Plate 2 shows four cylinders, each with identical surface properties but different sizes and orientations computed to a resolution of 256 by 256 pixels. The large central cylinder and the one on the lower right exhibit severe specular aliasing. The cylinder on the upper left is also exhibiting specular aliasing even though none is visible at present. For if it is moved slightly, aliased highlights will appear on its surface. This is also true of the cylinder in the upper right. Plate 3 shows the same scene rendered using the highlight clamping algorithm. The highlight down the central and lower right cylinders are now smooth with no jaggies present. A highlight is faintly visible on the cylinder in the upper left. Now, even if the cylinder is moved, no highlight will flicker on and off. The cylinder on the upper right has no highlight visible as it is too faint. There is so much curvature in this cylinder that any visible highlight would cause aliasing due to the severe undersampling. Plate 4 shows the same scene rendered with more aggressive clamping (99.9 percent power). The highlights are brighter and not as spread out and aliasing is not noticeable. Plate 5 shows a scene with two cones with the one on the right being highlight anti-aliased. The cone shape is useful in that it illustrates a continuous transition from low curvature to high. As can be seen, the transition into the high curvature region is smooth when anti-aliased.

Conclusion

We have introduced a simple analytic algorithm that, given the change of α within the pixel, detects when specular aliasing is present and indicates a sampling rate to overcome it. We have also introduced a very fast and simple algorithm that removes specular aliasing without increasing the sampling rate.

I would like to thank Xerox PARC, and especially Frank Crow, for providing the facilities and support for much of this research project. Also, thanks to NSERC for their continuing support.

References

Blinn, 1978.
J.F. Blinn, "Simulation of Wrinkled Surfaces," *Computer Graphics,* 12(3), pp. 286-292 (August 1978).

Bui T. Phong, 1975.
Bui T. Phong, "Illumination for Computer Generated Pictures," *Comm. of the ACM,* 18(6), pp. 311-317 (June 1975).

Carpenter, 1984.
L. Carpenter, "The A-buffer, an Antialiased Hidden Surface Method," *Computer Graphics,* 18(3), pp. 103-108 (July 1984).

Catmull, 1978.
E. Catmull, "A Hidden-Surface Algorithm with Anti-Aliasing," *Computer Graphics,* 12(3), pp. 6-10 (August 1978).

Cook, 1986.
R.L. Cook, "Stochastic Sampling in Computer Graphics," *Trans. on Graphics,* 5(1), pp. 51-72 (January 1986).

Crow, 1977.
F.C. Crow, "The Aliasing Problem in Computer-Generated Shaded Images," *Comm. of the ACM,* 20(11), pp. 799-805 (November 1977).

Crow, 1981.
F.C. Crow, "A Comparison of Antialiasing Techniques," *IEEE Computer Graphics and Applications,* 1(1), pp. 40-48 (January 1981).

Crow, 1982.
F.C. Crow, "Computational Issues in Rendering Anti-Aliased Detail," *IEEE 1982 Spring COMPCON,* pp. 238-244 (1982).

Duff, 1989.
T. Duff, "Polygon scan conversion by exact convolution," *in Raster Imaging and Digital Typography, Edited by J. Andre, R. Hirsh, Cambridge Univ. Press; Proceedings of RIDT89 Intl. Conf., Lausanne, Switzerland,* pp. 154-168 (October 1989).

Hwei P. Hsu, 1970.
Hwei P. Hsu, "Fourier Analysis," Simon and Schuster, N.Y. (1970).

Norton, 1982.
A. Norton, A.P. Rockwood, and P.T. Skolmoski, "Clamping: A Method of Antialiasing Textured Surfaces by Bandwidth Limiting in Object Space," *Computer Graphics,* 16(3), pp. 1-8 (July 1982).

Oberhettinger, 1973.
F. Oberhettinger, "FOURIER EXPANSIONS: a collection of formulas," Academic Press, N.Y. (1973).

Pratt, 1978.
W.K. Pratt, "Digital Image Processing," Wiley-Interscience (1978).

Saito, 1989.
T. Saito, M. Shinya, and T. Takahashi, "Highlighting Rounded Edges," *CG International '89* (1989).

Tanaka, 1990.
T. Tanaka and T. Takahashi, "Cross Scanline Algorithm," *Eurographics '90,* pp. 63-74 (1990).

Williams, 1983.
L. Williams, "Pyramidal Parametrics," *Computer Graphics,* 17(3), pp. 1-11 (July 1983).

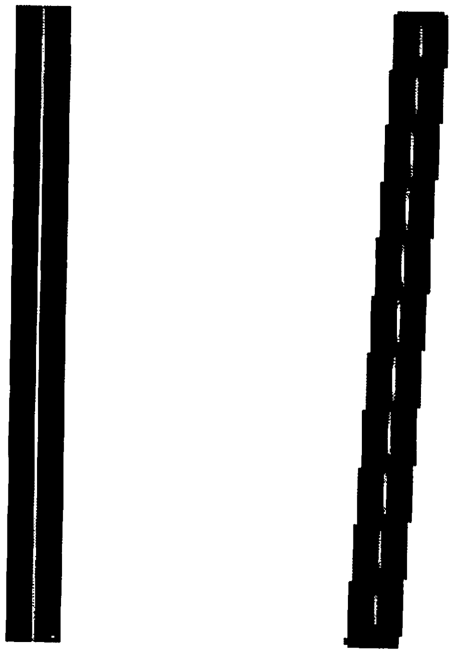


Plate 1

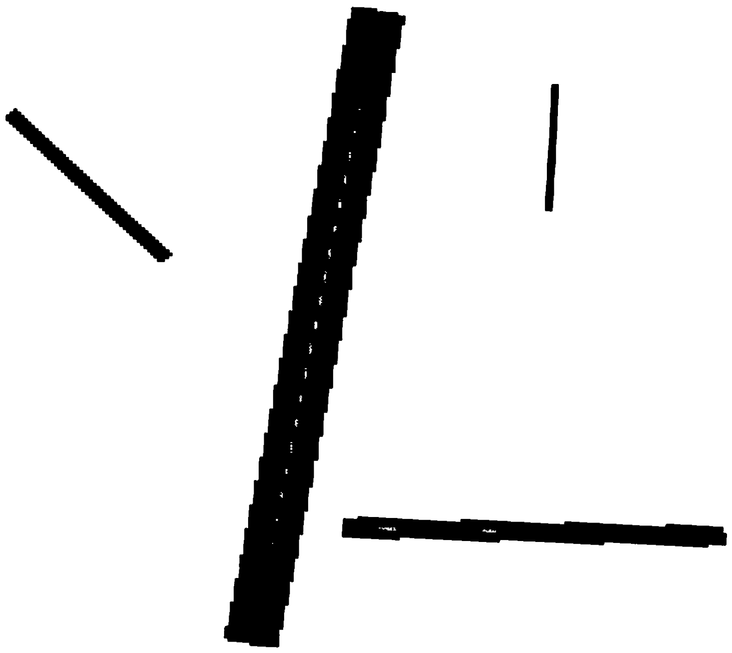


Plate 2

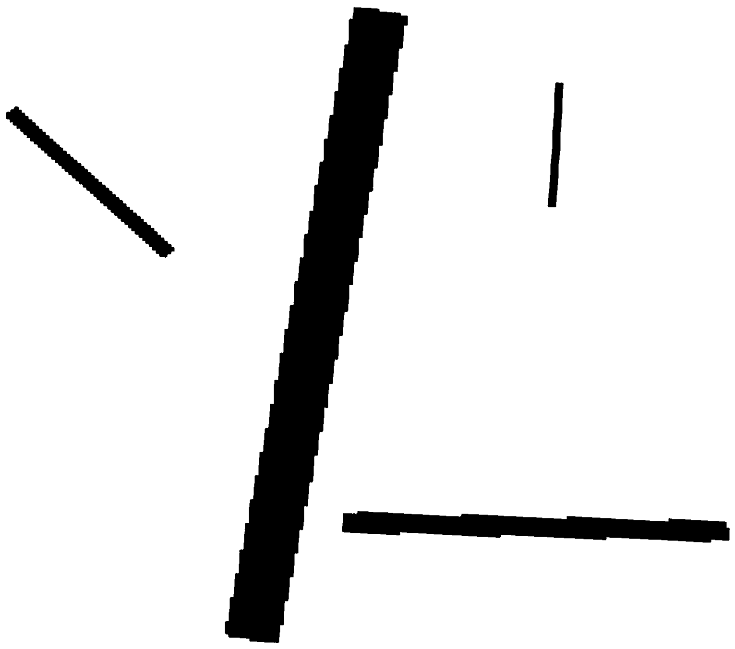


Plate 3

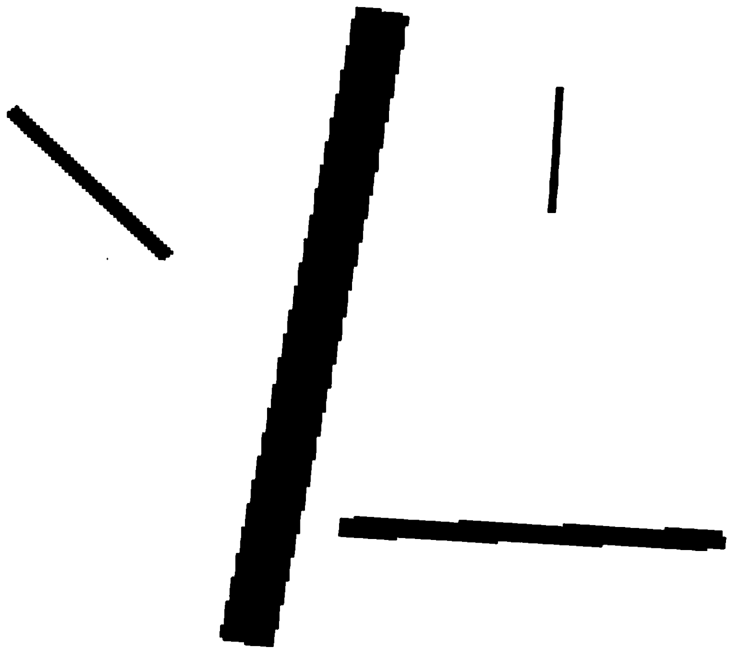


Plate 4

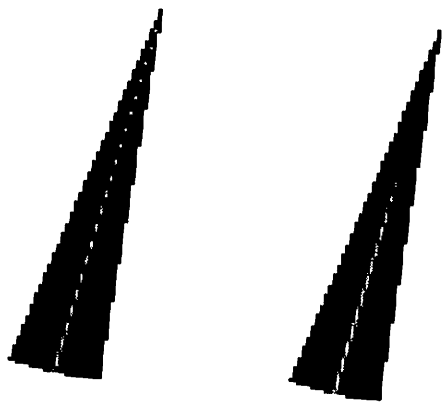


Plate 5

Hierarchical Poisson Disk Sampling Distributions

Michael McCool
Eugene Fiume

Dynamic Graphics Project
Computer Systems Research Institute
University of Toronto
6 King's College Road
Toronto, Ontario, Canada M5S 1A1
{mccool,elf}@dgp.utoronto.ca

Abstract

A relaxation method is presented that generates a series of spatially distributed stochastic samples following the Poisson disk distribution over a range of scales. The series is structured so that a prefix subsequence forms a lower frequency sampling pattern that is also *Poisson disk distributed. Unlike strict dart-throwing, the relaxation procedure is guaranteed to terminate. The hierarchical structure allows straightforward adaptive sampling, using techniques analogous to ordered dithering. Also presented is an efficient method to optimize the location of samples under a minimum root mean square quantization error criterion.*

Résumé

Une méthode de relaxation est présentée qui génére une série d'échantillons distribuée de façon aléatoire selon une distribution de disque Poisson pour une gamme d'échelles. La série est construite de façon à ce que les premiers échantillons correspondent aussi *à une distribution de disque Poisson, mais d'une fréquence plus basse. Contrairement à la méthode du lancé de darts, la methode presentée termine tourjours. La structure hierarchique permet un échantillonage adaptatif simplement en se servant d'une technique semblable à celle connu sous le nom de* ordered dithering. *Une methode efficace pour optimizer le placement des échantillons dans le sens du valeur quadratique moyenne est également presentée.*

Keywords: ray tracing, stochastic sampling, Poisson disk, adaptive sampling, dithering, optimization, antialiasing.

1 Introduction

Stochastic sampling has arisen as an important technique for the estimation of multidimensional integrals in computer graphics [Coo86b, Mit91, LRU85, Mit91]. Area integrals are implicit in the solution of such problems as antialiasing, soft shadows, and the simulation of finite camera apertures (causing depth-of-field effects). Integration over time is needed for motion blur. All these problems can be treated together, resulting in a large multidimensional domain of integration. Ray tracing in particular has benefited from this approach because of its inherent point sampling limitations.

Error bounds can be derived from the variability of the samples: variance, contrast, etc. An approximate error bound on the results of stochastic sampling using variance is given by

$$\epsilon \propto V\sqrt{\frac{s^2}{N}},$$

where N is the number of samples, s^2 is the variance of the sum, and V is the volume of the domain of integration [LRU85, PFTV89]. This error relationship is typically used in numerical applications of the Monte Carlo (stochastic sampling) method. Note, however, that it only decreases as a function of $\sqrt{N}$. An error bound based on contrast is given in [Mit87]; this should more closely match the psychophysical characteristics of the human visual system. Contrast is defined as

$$C = \frac{I_{\max} - I_{\min}}{I_{\max} + I_{\min}}.$$

These expressions are particularly useful in adaptive sampling techniques, where only enough samples are taken to satisfy an error bound in a local region. For the contrast measure, supersampling can be initiated if the contrast is above a certain bound, indicating high variability.

Research has focused on the problem of choosing sample positions so as to minimize the impact of this inevitable error while minimizing the number of samples. Both can be accomplished by carefully choosing the position of the random samples.

Instead of structured artifacts such as moiré patterns or staircasing ("jaggies"), aliasing error due to undersampling appears as noise. The distribution of random samples shapes the frequency spectrum of this noise.

Visually, low-frequency noise is particularly objectionable, appearing as "structure" in the image. Also, a low-pass reconstruction filter will typically be applied after the sampling, filtering out high-frequency noise; often this filter will just be a summation of the sampled values (a box filter). For both these reasons, the sampling strategy should try to concentrate the noise in high spatial frequencies.

Mitchell [Mit91] has recently studied the problem of choosing samples across all dimensions of the rendering problem to optimize the spectral characteristics of the noise on the resulting image. This study has some results in common with his; in particular, he gives an algorithm which will generate hierarchical distributions very close to the ones we give in Section 4, and can be used as an alternative to the algorithm given in that section. However, this paper also presents a new and very efficient approach to high-quality adaptive sampling using these distributions, and presents an efficient optimization algorithm that can be applied to other problems such as the quantization of normals [McC91] or generation of random meshes [Tur91].

We will focus on the sampling problem in two spatial dimensions, which applies to the subproblems of area light sources, antialiasing, and camera aperture simulation.

The problem of selecting a spatial sampling distribution has been studied in graphics [CPC84, DW85, Coo86b, Coo86a, CPC84, Mit87], dithering [Uli87], and coded aperture reconstruction [BS81]. The analysis in this paper will follow that in [Uli87]; we will analyse distributions empirically using frequency-domain techniques.

Totally random selection of points is not a good solution, as will be shown in Section 3.1. Other approaches to this problem have included jittering samples from a periodic grid, n-rooks, and the Poisson disk. Because of space restrictions, not all of these sampling approaches have been analyzed in this paper. The following sections will focus on optimizing the Poisson disk approach after first introducing some analysis tools.

There are indications that the Poisson disk distribution is one of the best from a spectral point of view. In comparison with jittering it is difficult and expensive to generate and use: the following analysis assumes a table-based implementation.

2 Analysis Tools

We need a set of tools to analyse the frequency domain characteristics of different sampling patterns. These tools will reduce the response to two one-dimensional graphs, radial power and radial anisotropy. Radial power will measure power at each spatial frequency, regardless of orientation; anisotropy will measure the variability of this measure about all orientations.

An empirical analysis will be used. It is often simpler to see how a sampling pattern actually behaves than to attempt to derive its response, since any derivation will require some simplifying assumptions that may limit the applicability of the solution. For example, the sequential generation of samples using dart throwing (Section 3.2) is usually not modelled.

We assume that radially symmetric responses are appropriate. Alternatives are possible, i.e. highpass two-dimensional filters with a diamond-shaped stopband. These should be simple extensions of the approach outlined here.

A distribution of samples can be represented by a set of impulses in the plane. To analyse the frequency domain response, the *periodogram* of the distribution is evaluated and measures of power and power variance in a set of circles about the centre are computed.

The periodogram is the Fourier transform of the autocorrelation function. The *autocorrelation* r_f of a real signal f is defined as

$$r_f(\vec{y}) = f(\vec{x}) \star f(-\vec{x}) = \int f(\vec{x}) f(\vec{x} - \vec{y})\, d\vec{x}$$

For real signals the Fourier transform of the autocorrelation function is equal to the square of the magnitude of the Fourier transform of $f(\vec{x})$:

$$R_f(\vec{\omega}) = \mathcal{F}[r_f(\vec{y})] = |\mathcal{F}[f(\vec{x})]|^2,$$

according to the complex identity $zz^* = |z|^2$, the convolution theorem, and the time-reversal theorem. Using the periodogram will allow the power and anisotropy at different frequencies to be evaluated, since we can compute the anisotropy at any radius. At points closer to the origin, energy in the periodogram corresponds to low-frequency energy in the autocorrelation, and therefore long-distance spatial correlation. Since the periodogram is also the spectrum squared, low frequency power also corresponds to low frequency, or large-scale, structure in the function $f(\vec{x})$.

We want to avoid low-frequency structure in our sampling patterns because this structure could be erroneously perceived as structure arising from the source image. The validity of this quality measure depends on the quality of the reconstruction filter, which should reject high frequencies. It should be noted that a box filter has a $\mathrm{sinc}(\omega_x)\mathrm{sinc}(\omega_y)$ frequency response, which allows some leakage at high frequencies and is consequently not the best choice.

To evaluate the periodogram of a distribution, it is necessary to find the mean periodogram of samples drawn from the distribution. In the following analysis, average periodograms are computed by summing together 100 periodograms of sample images drawn from each distribution. Each image has a resolution of 32×32 and contains 64 impulses, a density of 1/16. Images are defined over the unit square. An impulse is represented by a value of 1 at the appropriate location in the image. Note that the following approximations and assumptions are made in this analysis:

1. The impulses are finite-width.
2. The location of the impulses are rounded to the nearest grid location.

3. A finite-resolution grid is used.
4. The distribution on the unit square is assumed to tile the plane periodically.
5. A finite number of periodograms are used to estimate the mean periodogram.

Most of these approximations are made so that the discrete Fourier transform, in its efficient FFT form, can be used to evaluate the periodograms.

Note that the assumption that the distribution tiles the plane periodically is exactly the situation in many uses of the distribution, such as stochastic oversampling in ray tracing. The period of the pattern should be much larger than the pixel size.

Once the average periodogram R_f has been evaluated, we can reduce the information to two graphs by computing radial statistics. We define a set of annulli as in Figure 1. Within each annullus, the mean radial power P_i is computed as well as the variance s_i^2. This is the variance *within* an annullus, not between the samples used to create the average periodogram:

$$\begin{aligned} P_i &= \frac{1}{A_i}\int_0^{2\pi}\int_{f_i}^{f_{i+1}} R_f(f\cos\theta, f\sin\theta) f\, df\, d\theta \\ s_i^2 &= \frac{1}{A_i}\int_0^{2\pi}\int_{f_i}^{f_{i+1}} (R_f(f\cos\theta, f\sin\theta) - P_i)^2 f\, df\, d\theta \end{aligned}$$

where $A_i = \pi(f_{i+1}^2 - f_i^2)$ is the area of annullus i defined by radii f_i and f_{i+1}. In practice, the "area" is given by the number of samples on the discrete 32×32 grid that fall within the bounds of a given annullus. The sample size increases approximately linearly up to the edge of the square, then decreases for the annulli that have samples only in the corners of the periodogram. For this analysis, 20 equally-spaced annulli from the centre (0 frequency, or DC) to the corner (highest possible frequency) were used. Figure 2 shows the sample size at each frequency. A dotted line has been placed where the edge of the square is first encountered, and will be present on all diagrams in which it is relevant.

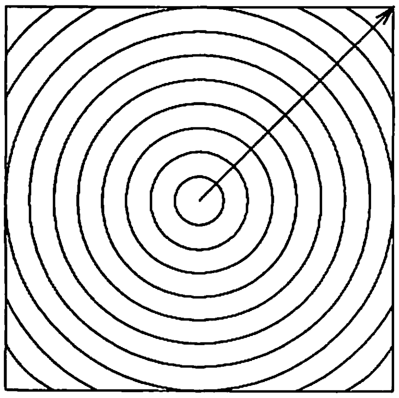


Figure 1: Annulli defined to average the periodogram radially. Twenty annulli were used for this analysis.

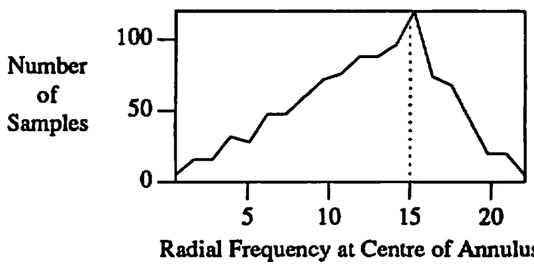


Figure 2: Sample size as a function of radial frequency for estimation of radial statistics in each annullus.

Two graphs will be shown for each distribution analysed: the mean radial power $P_r(f)$ and the anisotropy $A_r(f)$ defined as

$$A_r(f) = \frac{s^2(f)}{P_r^2(f)}.$$

Larger values of anisotropy indicate a greater uneveness in the radial distribution of power. Power and anisotropy vary over a wide range. For this reason they will be plotted in decibels; a value x is expressed in decibels according to the relation $x_{\mathrm{dB}} = -20\log_{10}(x)$. A doubling of power or anisotropy approximately corresponds to a 6dB change.

The periodograms will also be displayed in three-dimensional graphs. In these graphs, the DC peak has been reduced by a factor of 4 and actual power is shown, not decibels. These plots are for qualitative comparison of sampling distributions.

3 Random Distributions

Several random sampling patterns have been studied. The one most used in practice is jitter sampling, which moves samples off a uniform grid by a random amount. This technique is simple to implement, but is not spectrally optimal.

This section analyses only Poisson disk sampling and derivatives, which have a better frequency response than jitter (less low frequency noise). They also have another advantage over jitter sampling: an arbitrary number of samples may be used.

The following subsection first analyses strict random sampling for comparison purposes.

3.1 Poisson

The simplest random distribution is the so-called "Poisson" distribution. The coordinates of a point are selected at random from a uniform distribution. Every point is completely independent of the others, making the generation of points a Poisson process. The number of points in an area A is Poisson-distributed with mean equal to the area times the density. For this reason, the distribution is often called a Poisson distribution. This unfortunate terminology should not be confused with the univariate discrete Poisson distribution.

A sample two-dimensional Poisson distribution is shown in Figure 3. The tile outlined in the centre of the diagram tiles the plane periodically in a square array; parts of adjacent tiles are shown. All distributions in this study will be shown this way.

The centre tile shown here would extend over several pixels if this sampling pattern were being used for antialiasing.

The periodogram of this distribution, as well as an average of 100 other instances, is shown in Figure 4. Radial averages for these distribution are shown in Figure 5. Anisotropy is shown in Figure 6.

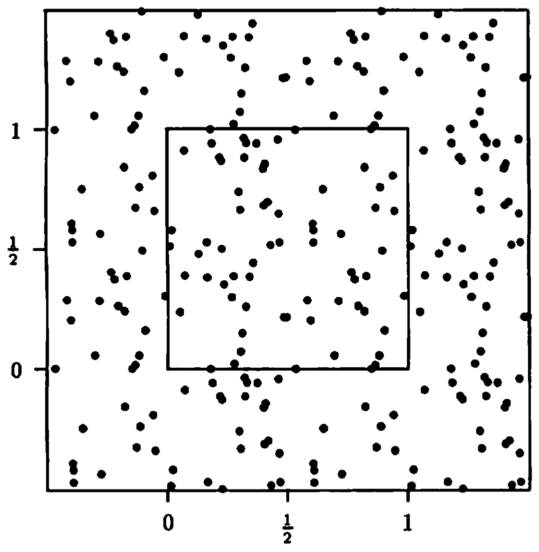


Figure 3: Uniform distribution of impulses on the unit square.

Note how clumpy the spatial distribution in Figure 3 appears; this is due to the image having significant power at all frequencies, as we can see in looking at the periodogram. The clumps arise because the low frequency power in the spectrum of the distribution allows a variation in the local average of intensity to occur. However, we see that the periodogram is a reasonably good approximation to an impulse function.

The distribution is seen to have an even amount of anisotropy over all frequencies, which is desirable for our application; we do not want any preferred direction in the noise. The clumpiness, however, will undoubtably give poor performance. To eliminate the clumpiness we will have to eliminate the low frequencies in the spectral response.

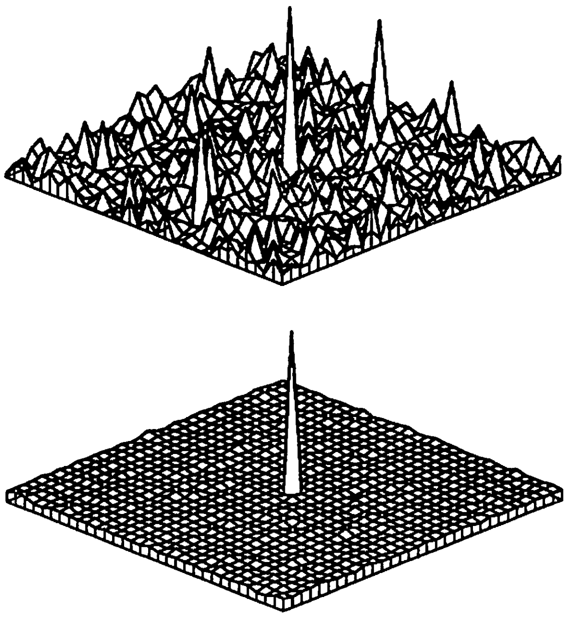
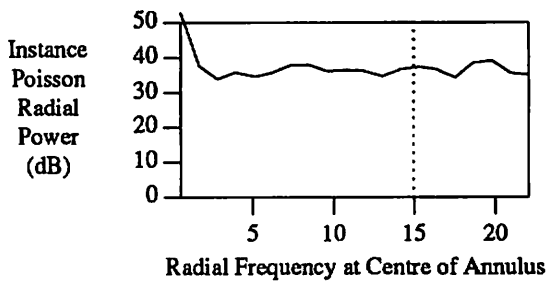


Figure 4: Sample and Average ($n = 100$) periodograms for the Poisson distribution. The DC peak has been reduced by a factor of 4.



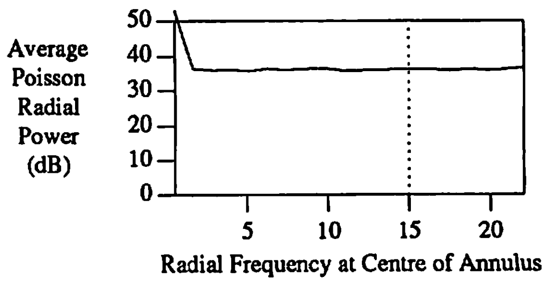
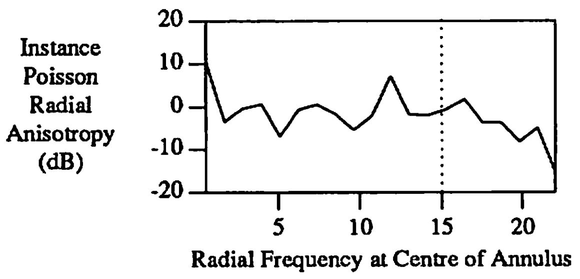


Figure 5: Radially averaged power for the spatial Poisson distribution. Top: single sample distribution; bottom: average ($n = 100$) radial power.



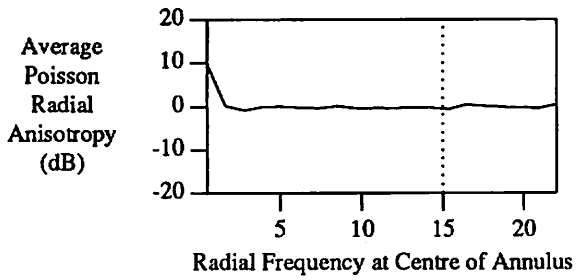


Figure 6: Radially averaged anisotropy for the spatial Poisson distribution. Top: single sample distribution; bottom: average ($n = 100$) anisotropy.

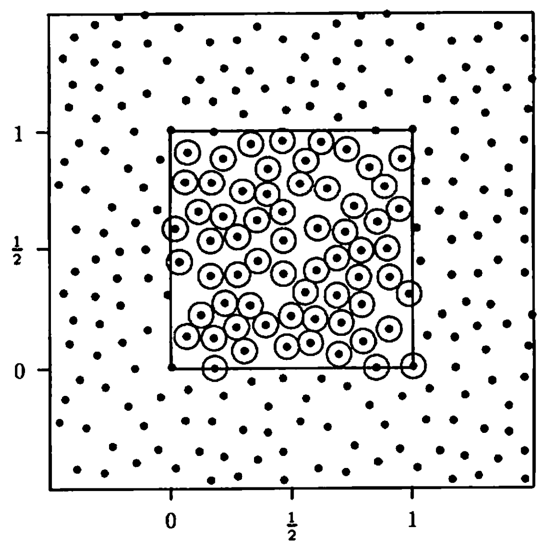


Figure 7: Poisson disk distribution in the plane. Center tile drawn with disks having the same diameter as the minimum intersample spacing.

3.2 Poisson Disk

The Poisson disk distribution is a Poisson distribution in which no two points are closer than a minimum distance[1].

In theory, to get a sample of a Poisson disk distribution one must generate Poisson distributions and evaluate each one according to the minimum distance requirement. This is not a very practical algorithm, and effective approximations have been developed.

The *dart-throwing* algorithm [Coo86b] places points sequentially. Each new point is compared to points already placed; if it is too close it is discarded. The algorithm terminates after a specific number of points have been placed, or it has proven impossible to place new points after a large number of attempts. A Poisson disk distribution in the plane, generated using dart-throwing, is shown in Figure 7. Circles are drawn about each point in the centre tile at half the minimum intersample spacing. A separation (disk diameter) of 0.1 that would result in a reasonable run-time was chosen experimentally. Intertile interference was also checked so that the periodic tiling of the plane would also satisfy the Poisson disk criterion.

Another class of algorithms that generate similar distributions are the *error diffusion* algorithms [FS75, Uli87]. These have been used in ray tracers [Mit87], but are limited to a discrete raster. It would also be possible to generate an adaptive sampling grid using an error diffusion algorithm on the grey-scale error image after the first pass of a ray tracer. However, error-diffusion dithering still has some artifacts, particularly in constant regions [Uli87]. We will not use an error-diffusion algorithm, but will use another technique adapted from *ordered* dithering in Section 5 when we discuss adaptive sampling.

It has been found that the Poisson disk distribution has desirable spectral properties, namely low energy for low frequencies. Sample and average periodograms for this distribution are shown in Figure 8. The radially-averaged statistics for this distribution are shown in Figure 9 and Figure 10.

4 Relaxation

Following a suggestion by Robert Lansdale (documented in [Lan91], and also referenced obliquely in [DW85]), the minimum radius for the distribution required by dart-throwing does not have to be fixed.

Points are placed starting with a large radius initially. Once no more space has been found at this radius for a certain (large) number of attempts, the radius is reduced by some fraction (which is just less than 1). The final distribution will still have a minimum separation given by the last radius used. A magnification fraction is also specified to increase the number of tests as the number of samples already placed increases. To save tests, the best-fitting sample during each iteration can be saved. Eventually, the minimum separation of this

[1]the Poisson disk distribution is characteristic of the receptors in the retina outside of the fovea [YJ83, WC83], where it prevents aliasing; the retinal image is subsampled in those regions. Within the fovea, the receptors are hexagonally packed at a density twice the highest frequency of the highest spatial frequency passed by the cornea/lens/iris system, so the Nyquist criterion is satisfied.

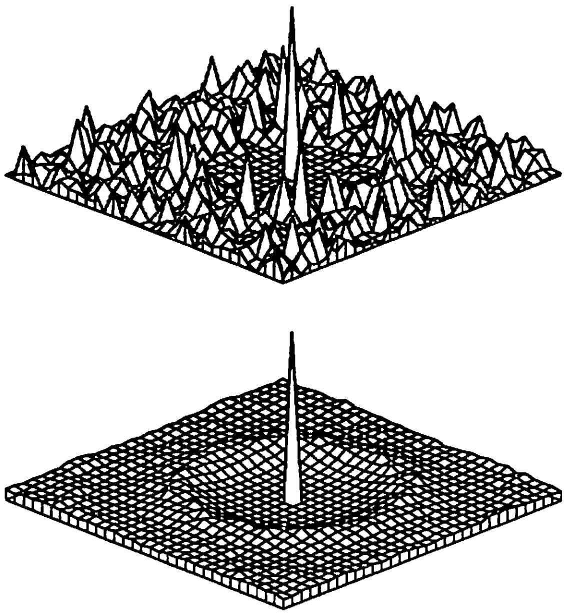


Figure 8: Sample and average ($n = 100$) periodograms for dart-throwing Poisson disk distribution. The central DC peak has been reduced by a factor of 4.

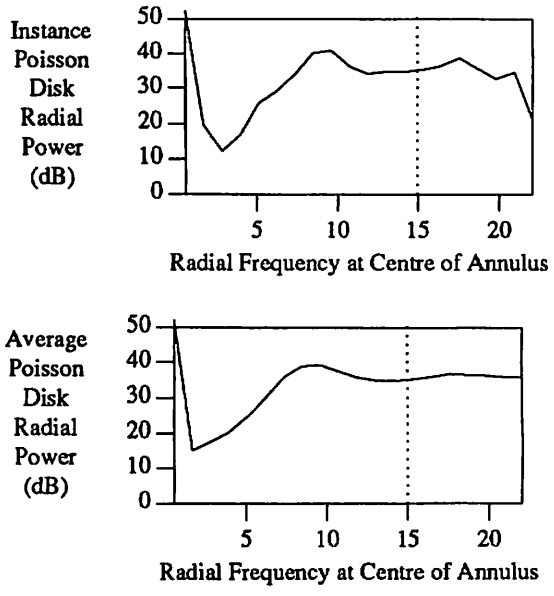


Figure 9: Radial average power for the Poisson disk distribution. Top: single sample distribution; bottom: average ($n = 100$) distribution.

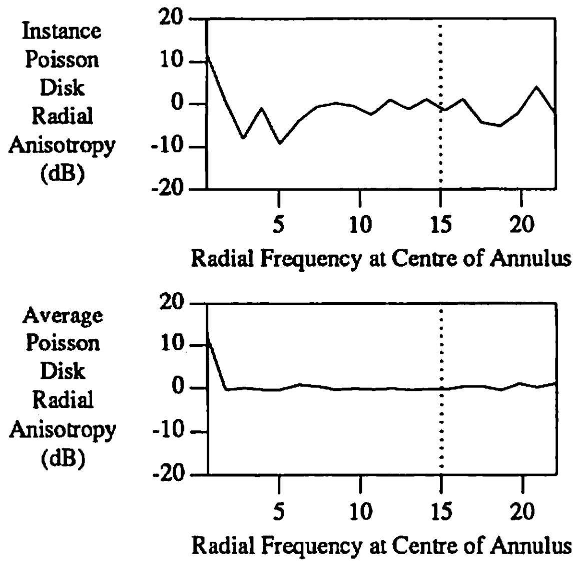


Figure 10: Anisotropy for the Poisson disk distribution. Top: single sample distribution; bottom: average ($n = 100$) distribution.

sample and the decaying radius will meet. In this way work from previous tests will not be wasted.

Such a distribution is shown in Figure 11. The circles surrounding each point in the centre tile indicate the radius in use when that point was placed. Visually, the distribution generated is similar to that generated by a fixed radius.

A similar approach was taken by [Mit91], but was not exploited for its adaptive sampling potential. Our algorithm relaxes the radius condition only when necessary, obtaining the best possible fit for the number of samples considered. Unfortunately, this may result in a longer running time. We will show empirically that the spectrum retains the desirable properties of the original Poisson disk distribution, at least over a limited range of scales. The algorithm differs from Mitchell's in two ways: a hard radius constraint is used; and the radii are guaranteed to be strictly decreasing (the last point could be addressed by sorting the samples generated by Mitchell's algorithm).

The new algorithm has many advantages. First, it will always terminate with a position for any desired number of samples, since the radius should eventually get small enough to allow every one to be placed. It is very possible that a dart-throwing algorithm with a fixed radius will not allow all samples to be placed, and will therefore fail to produce a distribution.

Second, this algorithm is less sensitive to its initial choice of parameters than strict dart-throwing. A larger threshold of trials or a larger decay fraction will increase running time and (hopefully) increase the "tightness" of the distribution, but will not cause the algorithm to fail totally. There might, however, be some regions which are more densely packed than others. This can

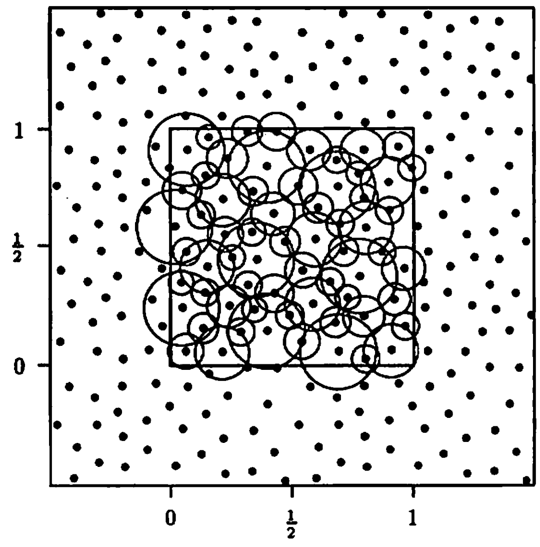


Figure 11: Approximation to a Poisson disk distribution generated by relaxation dart-throwing. The circles about points on the left indicate half the minimum intersample separation in use at the time the point was placed.

happen if the radius decay is too steep or the number of tests per iteration too few. Since the algorithm is adaptive, the same set of parameters can be used to generate distributions for a wide range of circumstances.

Finally, and probably most importantly, the resulting list of samples has a pyramidal property. Only one high-resolution sample distribution has to be generated for a wide range of sampling densities; if a lower sampling density is required, then only the initial points from the sample list are selected. These points will satisfy a larger minimum distance criterion than the full set.

In ray tracing, a common problem when using stochastic sampling is how to adaptively improve the sampling density. In the past, jitter sampling has been used rather than Poisson disk sampling because of the expense of generating each distribution. In [Mit91], Mitchell suggests scaling down the sample grid and replicating it; this, however, will introduce some unwanted periodicity which may compromise the effectiveness of the random grid and reintroduce coherent noise. This will only be a problem if a short period is used. With this algorithm, a single long list can be used and the required number of samples chosen, resulting in a consistent long period.

We need to analyze this distribution quantitatively. It will be shown that, empirically, its spectral properties closely match those of the fixed-radius dart-throwing algorithm. The distribution in Figure 11 was generated with an initial minimum spacing of 0.3 (disk radius of 0.15). The radius was multiplied by 0.99 after 1000 samples were tried without any success in placing a new point. Its periodogram is shown in Figure 12, along with an average periodogram of 100 other distributions generated with the same parameters. The radially-averaged statistics for this distribution are shown in Figure 13 and Figure 14.

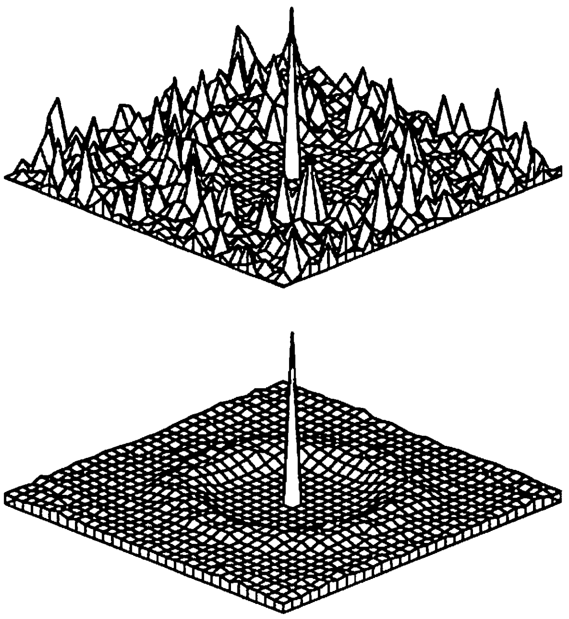


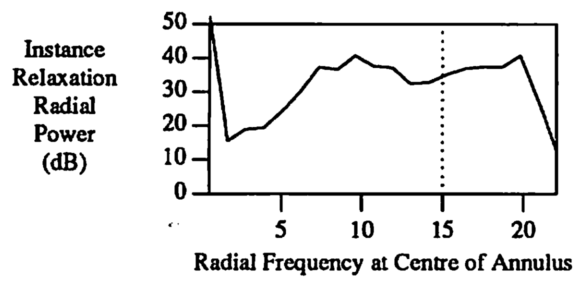
Figure 12: Periodograms for Poisson disk distributions generated by relaxation. Top: single sample distribution; bottom: average (n = 100) periodogram. The central DC peak has been reduced by a factor of 4.

The sequence of radii for the samples is given in Figure 15, as well as the average over the 100 tested distributions. Note that only the final samples approach the radius limit of 0.05 used for the basic dart-throwing in Section 3.2.

5 Adaptive Sampling

5.1 Continuously Variable Sampling Density

In many cases, it is desirable to locally modify the sampling density. For example, in the Introduction a formula was given for estimating the error, which depended on the variance of the samples collected. Selective supersampling corresponds to discrete levels of sample density, and can be triggered by a set of contrast thresholds. Using the pyramidal property of the relaxation-generated sequence, as many samples can be taken as needed, until an error criterion is satisfied. Every new sample will fit into a Poisson disk distribution, at ever-decreasing minimum spacing.



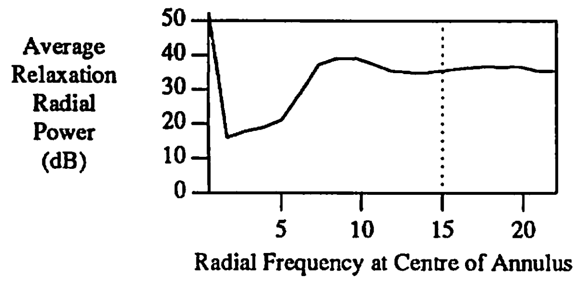
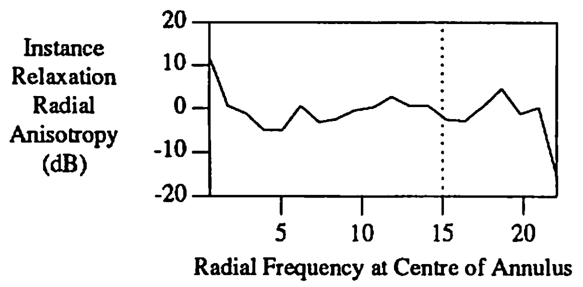


Figure 13: Radial average power for the Poisson disk distribution generated by relaxation. Top: single sample distribution; bottom: average (n = 100) distribution.



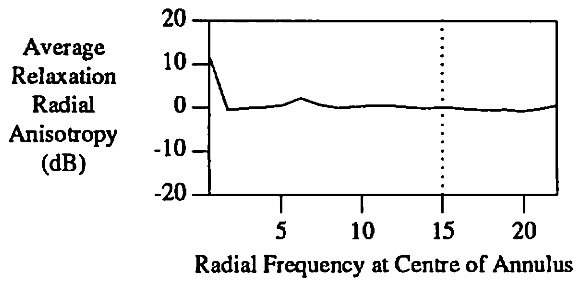
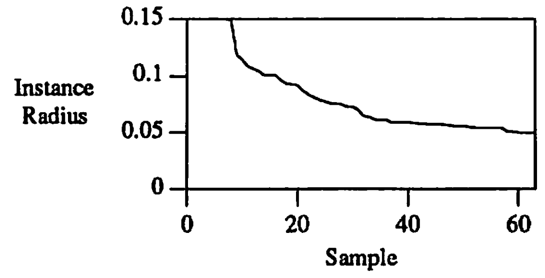


Figure 14: Anisotropy for the Poisson disk distribution generated by relaxation. Top: single sample distribution; bottom: average (n = 100) distribution.



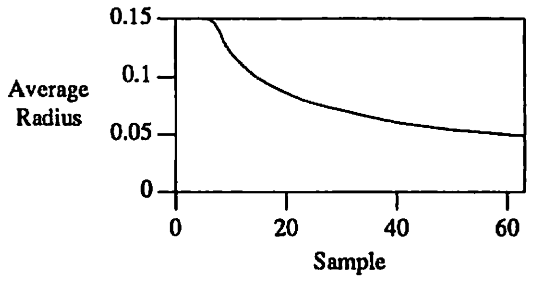


Figure 15: Sample Radii for the Poisson disk distribution generated by relaxation. Top: single sample distribution; bottom: average (n = 100) distribution.

A set of subsequences can be extracted and stored; each subsequence has samples that fall into predetermined partitions. The refinement can then be localized to only the partitions whose error estimates exceed a bound. The ultimate extension of this approach is importance sampling.

5.2 Importance Sampling

If the approximate shape or some factor of the function to be integrated is known, then the number of samples should be increased in areas in which the function has a high value. The sampling density then replaces a weighting factor. Many examples of this occur in computer graphics.

For example, consider the ray tracing simulation of a glossy surface. The energy reflected towards the eye is actually an integration over the hemisphere of the incoming energy, multiplied by the reflectance function. The reflectance function is known; the incoming energy is not. The integral can be performed using stratified Monte Carlo sampling. Random samples are shot in a distribution of orientations, but more samples should be shot in directions from which incoming energy will be strongly reflected.

For another example, it may be possible to replace a complex reconstruction filter with a simple box filter (mean of samples in a pixel) but to use importance sample according to the weight given by a higher-quality filter. This would have the effect of concentrating more samples in the center of the pixel.

Importance sampling can be trivially solved using an approach from ordered binary dithering. In ordered

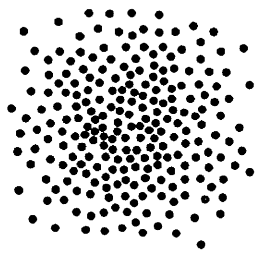


Figure 16: Adaptive Poisson disk sampling; density weighted by $\exp(-5(x^2+y^2))$.

dithering, a mask is used which is an array of thresholds, one per pixel. A full mask is usually generated by repeating some basic pattern. At any point, if the pixel in the source image is greater than the threshold at that point in the mask, then the pixel is turned on.

A similar idea can be used if we have previously generated a large Poisson disk distribution with uniform density using relaxation. We know that the more samples chosen from the list and displayed, the denser the samples will be, although they will all follow a Poisson disk distribution at some scale. Therefore, select the samples according to the following algorithm:

1. For each sample $\vec{s_i}$, compute its rank as $r(\vec{s_i}) = i/n$, $1 \leq i \leq n$.

2. Let
$$x(\vec{s_i}) = \frac{f(\vec{s_i}) - \min(f)}{\max(f) - \min(f)},$$
where f is the known importance function. In the first example above, this would be the value of the reflectance function; in the second, it would be the weight of the reconstruction filter.

3. If $r(\vec{s_i}) \leq x(\vec{s_i})$, choose $\vec{s_i}$ as an evaluation sample point; otherwise, discard it. If a sample is chosen, it is part of the pattern needed to represent the density x.

This is very efficient, since the same table of samples can be used over and over for a variety of importance functions, as long as care is taken to not allow overly periodic use of the same sampling pattern.

An example of a set of samples whose density is weighted by $\exp(-5(x^2+y^2))$ over $[-0.5, 0.5]^2$ is shown in Figure 16. A subsequence of 227 samples were selected out of 512.

Once a subsequence has been generated, it can *still* be used in the manner described in Section 5.1, since it will still satisfy the increasing density criteria. Only as many sample points are taken as are needed.

6 Optimization

Although a Poisson disk distribution has some nice qualities, it is possible to generate a better one from a quantization viewpoint.

We would like the distance from every point on the plane to the nearest sample point to be as small as possible. If the function to be estimated varies smoothly, then this strategy will result in the minimum root mean square (RMS) error. It will also mean that no large "gaps" will be left in the sampling structure.

We know the optimal solution to this problem, at least on the plane: a periodic distribution of samples in a hexagonal grid. However, we want to avoid periodicity. This should not be a problem, since optimization procedures that can find global optimums are rare. In this case, we actually *want* an optimization procedure that will return a somewhat suboptimal solution close to our initial configuration. We can then initialize with a Poisson disk distribution and hopefully derive a result which still retains most of its useful properties.

The Generalized Lloyd Algorithm (GLA) described further in [Llo82, JS86, For88] is based on a minimization of the (root) mean square error in quantizing each position in a multidimensional metric space to the nearest sample point. It is a strictly descending method and as such is unlikely to find global optimums in complex situations.

Given $\vec{x}$ distributed according to probability density function $p(\vec{x})$ over a space with distance metric $d(\vec{x}_1, \vec{x}_2)$, the mean square error is:

$$x_{\mathrm{MSE}} = \sum_{i=1}^{N} \int_{\mathcal{D}_i} p(\vec{x}) d^2(\vec{x}, \vec{x}_i)\, d\vec{x},$$

where $\mathcal{D}_i$ is the domain of $\vec{x}_i$: the set of all points that map to $\vec{x}_i$. Note that both probability and error are accounted for in this metric. This metric more heavily weights extremes of error, while not neglecting the mean error rate. The mean square metric is convenient analytically because many useful results can be proven using it.

Two results that are useful in the design of quantizers give necessary (but not sufficient) conditions for an optimal MSE quantizer. In an optimal MSE quantizer,

1. Borders between two domains will be equidistant from the respective quanta under the distance metric d.

2. Quanta will be placed at the mass centres of their domains, where the mass centre of a domain is defined as
$$\dot{x}_i = \frac{\int_{\mathcal{D}_i} \vec{x} p(\vec{x})\, d\vec{x}}{\int_{\mathcal{D}_i} p(\vec{x})\, d\vec{x}}$$
In other words, the quanta should be the average value of their domain.

These conditions are easily proven.

Condition 1: Every point x with a non-zero probability should be mapped to the quantum which it is closest to under the distance measure d. Obviously, if it was not, the error would be higher and the mean square error metric larger. The boundaries are therefore equidistant under d, since points on the boundary can choose either quanta with no penalty in the error metric. If this was not true of the boundary, then the points on this boundary could be moved to the lower error domain and the metric reduced.

Condition 2: The value $\int_{\mathcal{D}_i} p(x)d^2(x, x_i)\,dx$ is the *moment of inertia* of $p(\mathcal{D}_i)$ around the point x_i; it obtains its minimum when x_i is $\hat{x}_i$, the mass centre of $p(\mathcal{D}_i)$.

The same results hold if the distance measure is angle on the surface of a sphere, with appropriate modifications to definitions (the edges of the domains will be great circles, for example).

The GLA alternates attempting to satisfy the above conditions. First, a set of initial quanta are chosen. Boundaries between domains are set using condition 1, and then the mass centres of each domain are computed. The quanta are moved to the mass centres to satisfy condition 2, which of course changes the boundaries and invalidates condition 1. The process is repeated until no more changes are necessary, assuming the $\vec{x}_i$ converge.

The optimization procedure can be used to generate a non-uniform distribution; simply vary the $p(\vec{x})$ term when computing the centroid. The probability distribution $p(\vec{x})$ should be smooth or the optimization may get wedged into a local minimum.

This technique will converge faster if the initial distribution is nonuniform. Such a distribution can be generated by the algorithm in Section 5. Alternatively, relaxation can be used directly, but the radius modified by the weighting function.

The positions of the quanta and the domain boundaries may not converge and may in fact go to infinity, depending on the shape of the probability distribution. This problem is avoided in the current problem by optimizing on a finite domain; the sides of the rectangle wrap around and so the optimization effectively takes place on the surface of a toroid.

Each iteration can be implemented by constructing a Voronoi diagram and finding the centre of mass of each cell. Construction of each diagram takes $O(n \log n)$ time, although in a sophisticated optimization procedure the diagram could be built incrementally; only small changes are required for each new iteration.

To generate the results shown here, the Monte Carlo method was used to estimate the centroid of each cell. This is an inefficient method compared to the Voronoi approach, but is simple to implement.

In other uses of this optimization scheme, a globally optimal solution may be important. Even if the positions of the quanta do converge, there is no guarantee that the quantizer thus derived will be globally optimal. In fact it is unlikely in complex situations, since the method is descending. If an optimal solution is desired, decaying noise can be added to simulate annealing. This will not be too expensive because the underlying optimization procedure is accurate.

A distribution optimized using a uniform probability distribution is shown in Figure 17. Ten iterations were used. As with the random distributions, 100 of these distributions were analysed and the results appear in Figures 18, 19 and 20. The average spectra shown in Figure 18 displays peaks corresponding to the horizontal and vertical directions. The danger of using this technique is that the optimization may actually find a global optimum and regain periodicity. An optimized distribution might be combined with jitter to avoid this problem.

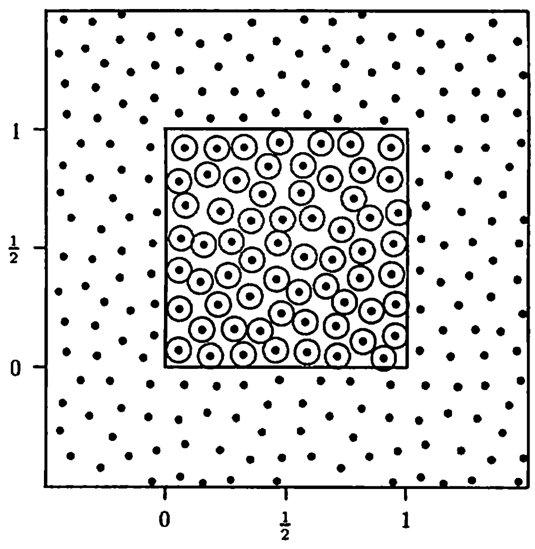


Figure 17: Optimization of a uniform distribution generated by dart-throwing. The circles about points on the central tile indicate half the minimum intersample separation in use at the time the point was placed.

7 Conclusions

A number of refinements in the generation and use of the Poisson disk stochastic sampling strategy have been presented and analysed empirically. It has been shown that by using relaxation, a sequence of samples can be generated such that prefix subsequences are also Poisson disk distributed. Sequences with this property can be used in both adaptive and importance sampling in a very efficient manner.

Unfortunately, to use these distributions in practice still requires large lookup tables, since the distributions were generated using a refinement of dart-throwing; the generation process is still relatively expensive and should not be included within the rendering loop.

An optimization technique has been presented which has uses in sampling and quantization. It will also be useful in other areas where points have to be placed uniformly within a finite domain, such as mesh generation over arbitrary smooth shapes. It can optimize sample positions to fit an arbitrary density distribution, although a smooth distribution has more chance of success.

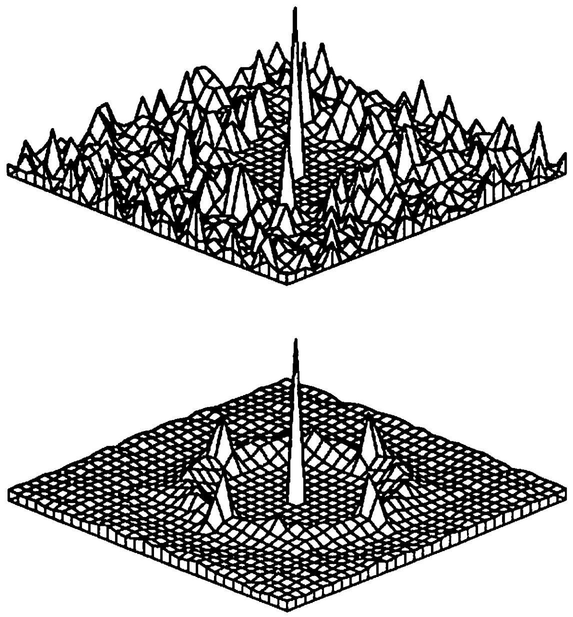


Figure 18: Periodograms for optimized distributions. Top: single sample distribution; bottom: average (n = 100) periodogram. The central DC peak has been reduced by a factor of 4.

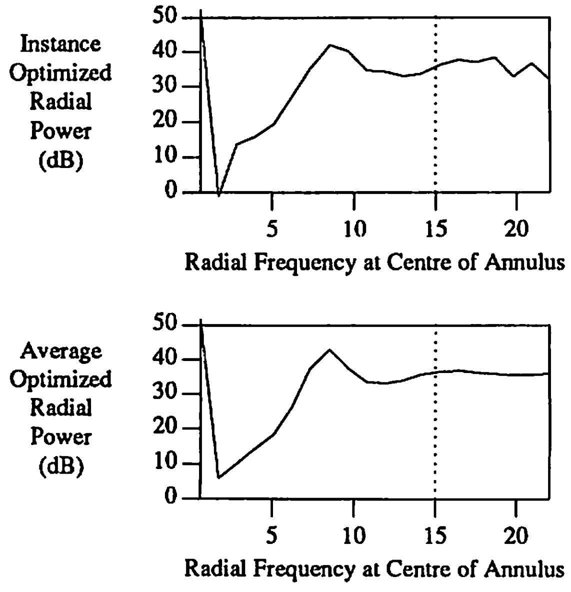


Figure 19: Radial average power for the optimized distributions. Top: single sample distribution; bottom: average (n = 100) distribution.

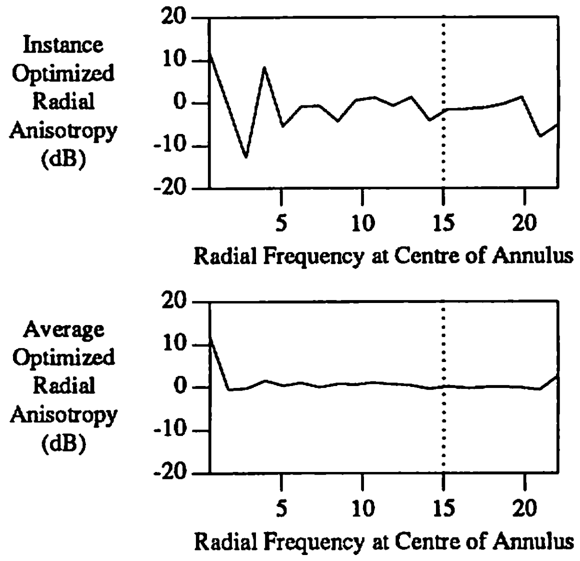


Figure 20: Anisotropy for the optimized distributions. Top: single sample distribution; bottom: average (n = 100) distribution.

8 Acknowledgements

An idea by Robert Lansdale helped to inspire this work, and the careful reading and comments by the anonymous reviewers were most helpful.

The long-term financial support of the Natural Sciences and Engineering Research Council of Canada and the Information Technology Research Centre of Ontario is gratefully acknowledged. Our graphics lab has also benefited from the generous financial support of Alias Research, Apple Computer, DEC and Xerox Corp.

References

[BS81] Harrison H. Barrett and William Swindell. *Radiological Imaging*, volume 2. Academic Press, 1981.

[Coo86a] Robert L. Cook. Practical aspects of distributed ray tracing. In *ACM SIGGRAPH '86 Developments in Ray Tracing seminar notes*, August 1986.

[Coo86b] Robert L. Cook. Stochastic sampling in computer graphics. *ACM Transactions on Graphics*, 5(1):51–72, January 1986.

[CPC84] Robert L. Cook, Thomas Porter, and Loren Carpenter. Distributed ray tracing. *ACM Computer Graphics (ACM SIGGRAPH '84 Proceedings)*, 18(3):137–145, July 1984.

[DW85] Mark A. Z. Dippé and Erling Henry Wold. Antialiasing through stochastic sampling. *ACM Computer Graphics (ACM SIG-*

GRAPH '85 Proceedings), 19(3):69–78, July 1985.

[For88] Bruno Forte. Topics in information theory: Lecture notes, April 1988. School for Advanced Studies in Industrial and Applied Mathematics, Department of Applied Mathematics, University of Waterloo.

[FS75] R. Floyd and L. Steinberg. An adaptive algorithm for spatial grey scale. *Society for Information Display (SID) International Symposium—Digest of Technical Papers*, pages 36–37, 1975.

[JS86] Neil Judell and Louis Scharf. A simple derivation of Lloyd's classical result for the optimum scalar quantizer. *IEEE Transactions on Information Theory*, IT-32(2), March 1986.

[Lan91] Robert Lansdale. Applications of 2D and 3D texture mapping in computer graphics. Master's thesis, University of Toronto, Department of Electrical Engineering, 1991.

[Llo82] S. P. Lloyd. Least squares quantization in PCM. *IEEE Transactions on Information Theory*, IT-28(2):129–136, March 1982. reprint of paper presented in 1957 at Inst. of Math. Stat. Mtg. in Atlantic City.

[LRU85] M. Lee, R. A. Redner, and S. P. Uselton. Statistically optimized sampling for distributed ray tracing. *ACM Computer Graphics (ACM SIGGRAPH '85 Proceedings)*, 19(3):61–67, July 1985.

[McC91] Michael McCool. Compact data structures for volume visualization. Master's thesis, University of Toronto, Department of Computer Science, 1991.

[Mit87] Don P. Mitchell. Generating antialiased images at low sampling densities. *ACM Computer Graphics (ACM SIGGRAPH '87 Proceedings)*, 21(4):65–72, July 1987.

[Mit91] Don P. Mitchell. Spectrally optimal sampling for distribution ray tracing. *ACM Computer Graphics (ACM SIGGRAPH '91 Proceedings)*, 25(4):157–164, July 1991.

[PFTV89] William H. Press, Brian P. Flannery, Saul A. Teukolsky, and William T. Vetterling. *Numerical Recipes*. Cambridge University Press, 1989.

[Tur91] Greg Turk. Generating textures on arbitrary surfaces using reaction-diffusion. *ACM Computer Graphics (ACM SIGGRAPH '91 Proceedings)*, 25(4):289–298, 1991.

[Uli87] Robert Ulichney. *Digital Halftoning*. MIT Press, 1987.

[WC83] D. R. Williams and R. Collier. Consequences of spatial sampling by a human photoreceptor. *Science*, 221(4608):385–387, 22 July 1983.

[YJ83] John I. Yellot Jr. Spectral consequences of photoreceptor sampling in the rhesus retina. *Science*, 221(4608):382–385, 22 July 1983.

Performing In-place Affine Transformations in Constant Space

Ken Fishkin
Xerox PARC
3333 Coyote Hill Road
Palo Alto, CA 94034 USA

Abstract

Affine transformations of 2-D frame buffer images are a common computer graphics operation. Such transformations take a rectangular raster of image memory, perform some affine transformation (e.g. scale, shift, shear, rotate) upon it, and write the result into some other rectangular raster of image memory.

If the source and destination share the same memory, the operation is termed in-place. Previous in-place affine transformation algorithms on an *m* by *n* region required O(max(*m*,*n*)) space for internal buffers. The algorithm presented here requires O(1) (constant) space: this allows in-place affine transformations to be performed on large images on processors with small memory.

Keywords: affine transformations, Catmull-Smith, frame buffer algorithms.

1. Introduction

An affine transformation is a transformation of the form

$$x' = Ax + By + C,$$

$$y' = Dx + Ey + F,$$

for arbitrary real values $A,B,C,D,E,$ and F.

Rotations, scales, shifts, and shears are all affine transformations. A computer graphics image is commonly considered as a rectangle whose contents are an array of pixels. An affine transformation can therefore be defined upon an image by performing the transformation upon the rectangle and then resampling [Catmull80].

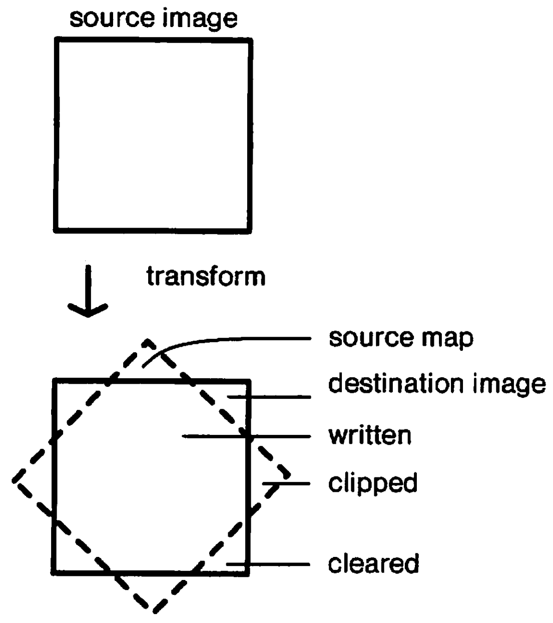


Figure 1: An affine transformation

2. Previous Work

2.1 The naive algorithm

A general affine transformation can be most easily implemented by creating a temporary image the same size as the source, and computing the value at each destination pixel in this temporary buffer. After all values have been computed, the temporary image over-writes the source image and is then freed. This approach requires O(*m n*) space when presented with an *m* by *n* source, and is only presented for comparison: it is not used in practice.

2.2 The Catmull-Smith algorithm

The popular Catmull-Smith algorithm [Catmull80] works by decomposing the affine transformation into two perpendicular shears. In the first shear the x' are computed, while the y coordinates are left untouched. In the second shear, the y' values are computed. To minimize artifacts the order of the two passes may be reversed, and transpositions or reflections may be required.

Each shear can be performed a scanline at a time, as y is unchanged by the x shear and x is unchanged by the y shear. Therefore, to transform an m by n image, a temporary scanline buffer of size $O(\max(m,n))$ is required.

3. Motivation for the New Algorithm

The Catmull-Smith algorithm has become a fundamental and accepted part of computer graphics software environments, with only minor extensions [Fraser85, Smith87] in the twelve years since its original publication. Of what interest is a new algorithm based upon it? There are both theoretical and practical motivations for the new algorithm.

> *Theoretical*: Just as the Catmull-Smith algorithm was motivated by the inability to process an entire image in-core, this paper presents an algorithm which is motivated by the inability to process an entire scanline in-core. An extension to a common $O(n)$ algorithm which uses $O(1)$ space may be of some theoretical interest.
>
> *Practical*: As computer graphics matures, it has begun to tackle more sophisticated problem areas with less specialized processors. These new problem areas, such as pre-press and satellite imaging, often have images with thousands or tens of thousands of pixels per line, and with tens or hundreds of bits per pixel. Further, the applications which manipulate these images often must run on processors with limited address space, such as the IBM PC [Microsoft87], the Macintosh [Apple85], or the Pixar Image Computer [Levinthal84]. In such environments, an algorithm such as the one presented is a necessity, not a luxury.

If both the image and the calculation buffer are stored in core, than an $O(1)$ algorithm is only of interest in the rare case where memory is just barely big enough to fit one image. An $O(1)$ algorithm is of more use when the image is kept in secondary storage (disk, frame buffer, etc.) and only the calculation buffer is stored in-core. In that case, the difference between an $O(n)$ and $O(1)$ buffer can be significant.

3.1 Software Architecture

The new algorithm analyzes the transformation and decomposes it into a series of calls to the Catmull-Smith algorithm. This allows the new algorithm to be incorporated as a extension to an existing library: no change in base software is necessary. Furthermore, any efficiencies or optimizations encoded into implementations of the Catmull-Smith algorithm can still be called upon: the wheel need not be re-invented. This does require, however, that the new algorithm issue "requests" for affine transformations which always transform rectangular sources into rectangular destinations, as this is the common format expected by software libraries.

The algorithm presented in sections 6-11, therefore, satisfies two constraints besides the $O(1)$ constraint. First, that the Catmull-Smith code is sacrosanct, and may not be changed or modified. This may be the case if that algorithm is provided in firmware, is written in micro-code, or is directly supported in hardware. Secondly, that the overhead involved in invoking that code is sufficiently significant that the number of calls to it should be minimized. If neither of those two constraints apply, then the simple and general algorithm presented in section 11 is sufficient.

4. Notation

The *source image* is the rectangular array of pixels comprising the picture which is to be transformed. The *destination image* is the rectangular array of pixels which is to receive the transformation. An *in-place* transformation is one in which the source image equals the destination image. The *source map* is the parallelogram which is defined by the map of the affine transformation over the source image. The source map and destination image will often be quite different. Pixels may be in the image but not in the map, in which case they are to be cleared to some background value. Pixels may be in the map but not in the image, in which case they are clipped. See figure 1.

Without loss of generality, we focus on the work to be performed by the x shear, where the shear acts on the x coordinates of the image without displacing it vertically. Let *iw* and *ih* be the width and height, respectively, of the source image. Let *dw* and *dh* be the width and height, respectively, of the destination image.

In the equation $x' = Ax + By + C$, we will rename A as *scale*, B as *tilt*, and C as *offset*, yielding $x' = scale * x + tilt * y + offset$.

Define $dst(x,y)$ as the x component in destination space of the image under the affine transformation of the given x,y coordinate in source space. Similarly, let $src(x',y')$ be defined as the x component in source space of the pre-image of the given x',y' coordinate in destination space. Note that $src(dst(x,y),y) = x$. Both the *dst* and *src* functions return a single x value, with no y value; they both map from R^2 to R^1.

For example, a transformation which scales the input image up by 10% and shifts it left by y pixels on the y'th scanline would have

$$x' = dst(x,y) = 1.1x - y, \text{ and}$$

$$x = src(x',y) = (x' + y) / 1.1$$

We also assume the function *bbox*(), which computes the bounding box in source space of the pre-image of the current destination rectangle, adding the appropriate amount at each end for filtering. Further, let M be the maximum size of a scanline in core, and let f be the amount which must be added to each end of a source scanline in order for a destination scanline to be calculated. The value of f is a function of the filter width, the particular filtering algorithm used, and the scale. When resampling, each source pixel influences source space for f pixels to the left and the right, yielding a total "penumbra" of $2f + 1$ pixels in reconstructed source space. When scaling down ($scale < 1$), this penumbra is spread over a great many destination pixels. Therefore, to ensure that at least 1 pixel can always be written, we require that

$$M >= (2f + 1) / \mathrm{MIN}(scale,1)$$

iw	source image width
ih	source image height
dw	destination width
dh	destination height
off	x' = off + x *scale + y * tilt
bbox()	bounding box
dst()	map from source to destination space
src()	map from destination to source space
M	pixel bandwidth limit
f	source padding due to filtering

Table 1: Notation

5. Why is this hard? The feedback problem

What makes an O(1) solution difficult? If no more than M pixels can be read or written at a time, one might imagine calling the Catmull-Smith algorithm on each M-sized chunk of the input scanline, writing the results out a piece at a time. The problem with this approach is that when an output piece is written, it may well overwrite a future input piece. This problem, on a grander scale, was exactly why the Catmull-Smith algorithm took pains to ensure that y is unchanging in the x pass, and why the naive algorithm allocated a huge buffer: *dst*(x,y) may be less than x, equal to x, or greater than x, and this relation may vary even within a scanline. The problem is exacerbated by the fact that anti-aliasing requires that an entire neighborhood of source pixels be readable when computing a single destination pixel.

Inefficiency is a further limitation of a per-line approach. The new algorithm will work by means of calls to the Catmull-Smith algorithm, which works on rectangular regions. It is a waste of inter-line coherence to call it on scanlines (or worse yet, parts of scanlines) only: we wish to call it as few times as possible, on regions as large as possible.

The problem is therefore to divide the original region, which is too large to transform all at once, into a set of regions, such that no pixel in any source region is over-written before it is no longer needed, and the number of regions is as few as possible.

6. An outline of the algorithm

It is advantageous to deal with affine transformations in what we define as *standard form*:

$$scale >= tilt, \text{ and } scale > 0$$

This form can be obtained by transposition (to satisfy the first clause) and reflection (to satisfy the second). Standard form allows a few more invariant assumptions: that increasing x increases x', and that vertical inter-scanline coherence is greater than horizontal inter-pixel coherence.

The algorithm consists of a series of transformation algorithms, with a case analysis to decide which transformation algorithm to perform. The transformation algorithms employed are more and more general and take more and more time: the case analysis finds the least general (and hence quickest) transformation algorithm appropriate for the particular transformation. The case analysis begins by dividing the source image into between 1 and 3 sub-images, depending on the characteristics of the transform. This process is described in section 8. At most one of these sub-images will require further case analysis, described in section 9. Some of that sub-images sub-images may require even further case analysis, described in sections 10 and 11. This completes all cases.

This case analysis is performed purely for optimization reasons: increasingly more difficult cases are processed by increasingly more general (but slow) algorithms. One could avoid all case analysis by using only the algorithm of section 11, but that could be far slower.

The case analysis routines all assume a subroutine, termed Helper, which performs an affine transformation on an arbitrarily large source and destination image, given the maximum internal scanline width M, and an evaluation order: left-to-right vs. right-to-left, and bottom-to-top vs. top-to-bottom. Before describing the case analysis in detail, we first describe 'Helper', the foundation of the system.

7. The Helper subroutine

The Helper subroutine is used to perform a piece-wise affine transformation from a given source image into a given destination image. It assumes that it need not worry about feedback. The Helper subroutine is solely concerned with slicing up the transformation into manageable chunks.

The obvious way to perform the Helper subroutine is to march along the source, transforming rectangles in the source into parallelograms which would then be written into the destination. This method was not chosen because, as Catmulll [Catmull80] says, "[n]ot only is this inconvenient, it is also difficult to prevent aliasing errors".

Instead, the Helper subroutines marches along in *destination* space: each destination rectangle is inverse-transformed to obtain a parallelogram in source space. That parallelogram is then rounded out into a rectangular bounding box (padding by f for filtering), and the Catmull-Smith algorithm is called. Since the source bounding box may be significantly larger than the source parallelogram, the shear may try to write a set of pixels which lie outside the current destination rectangle. However, since the destination rectangle lies along exact pixel boundaries, simple clipping will reject these extra pixels. See Figure 2.

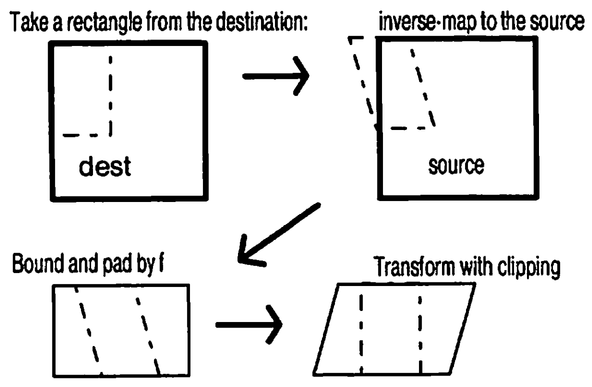

Take a rectangle from the destination:
inverse-map to the source
dest
source
Bound and pad by f
Transform with clipping

Figure 2: the Helper subroutine

What should be the dimensions of the destination rectangle? We wish to make it as large as possible without exceeding the M-pixel wide bottleneck.

Suppose we decide on a destination rectangle of size dx by dy. Then the constructed source rectangle will have width

$$sx = ((dx + (dy - 1)\,|tilt|)/scale) + 2f$$

Either dx or sx will be bounded by M. When $dx > M$, let $dx = M$ and $dy = dh$. When $sx > M$, there are more pixels to read than to write,

$$dx < sx,$$
$$dx < ((dx + (dy - 1)\,|tilt|)/scale) + 2f$$

We wish to minimize the number of destination rectangles, and therefore maximize the area of each. The problem is now to maximize $dx * dy$, where

$$1 <= dx <= M$$
$$scale < (dx + (dy - 1)|tilt|) / (dx - 2f)$$
$$((dx + (dy - 1)|tilt|)/scale) <= M$$

This is a quadratic programming problem. We approximate it by maximizing dy (setting it to dh), and then solving for a putative dx. If $dx < dy$ and $dx < dw$, then the putative rectangle is tall and skinny. To process an area closer to a square (this is desirable since Catmull-Smith has a per-scanline overhead), we recursively subdivide the rectangle by splitting it in two in y.

8. The first level of analysis

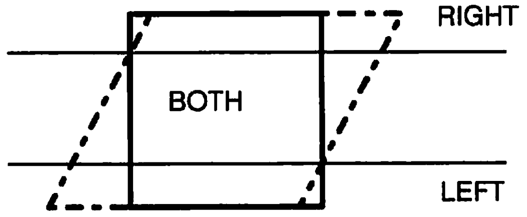


Figure 3: Different types of shear

The source image is subdivided into between one and three source sub-images, such that the map of the shear within each sub-image either:

1) overlaps it on the *left* on every line (type LEFT).
2) overlaps it on the *right* on every line (type RIGHT).
3) overlaps it on both the left *and* the right on every line (type BOTH).
4) overlaps it on *neither* the left *or* the right on every line (type NEITHER).

The dashed lines in figure 3 show the subdivision lines for that shear: the top region is of type RIGHT, the middle region is of type BOTH, and the bottom region is of type

LEFT.

8.1 Type LEFT

This case is easily processed. The Helper subroutine can be called on this region with an evaluation order of left-to-right.

8.2 Type RIGHT

The Helper subroutine can be called on this region with an evaluation order of right-to-left.

8.3 Type NEITHER

In this case, the transformation is scaling the source image down, and there is not a very pronounced tilt. However, evaluation order is still not clear: if *offset* (the translation component) is sufficiently large, then the first source pixels lie in the middle of the scanline: either right-to-left or left-to-right evaluations can fail. Therefore, this case is processed by decomposition into two shears:

1) The Helper subroutine is called to perform the scale component of the shear. Without any translation, *dst*(x,y) < x for all x, and so the algorithm should proceed left-to-right.
2) The Helper subroutine is called again to perform the non-scale component. If *offset* <= 0, then *dst*(x,y) < x, and so the algorithm should proceed left-to-right. Otherwise, it should proceed right-to-left.

8.4 Type BOTH

This case is complex and treated in detail in the next section.

9. The second level of analysis: type BOTH

Using the case analysis of section 8, 3 of the 4 possible shear configurations could be processed. This section focuses on the fourth, most difficult case, when the source map overflows the destination image on both sides. This case is more difficult because every pixel in the destination must be written, and hence every pixel in the source is at risk of being over-written before it is read.

Checks are first made to see if the shear falls into one of two special cases:

9.1 small-source

Suppose that the shear is a large scale-up, with little or no translation/tilt component. In this case, only a small part of the source is necessary. The algorithm can "cut out" that portion, and use the other portions as it wishes. The area of the source needed is considered "small" if and only if

$$bbox() <= dw$$

This means that the area of the source needed to compute the destination (*bbox*()) fits entirely within the destination (width *dw*). See Figure 4 for an example.

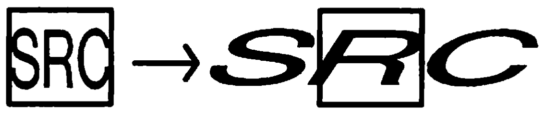


Figure 4: small-source: only the 'R' part of the source is needed

9.2 small-dest

Conversely, suppose that the shear is a large scale-down. Regardless of the translation/tilt component, in this case only a small number of destination pixels are truly "derived": the others will be set to a certain background color. In this case the algorithm can compute that portion and then shift/shear it into place. The area of the destination needed is considered "small" if and only if

$$iw * scale <= dw$$

This means that the width of the scaled source image (*iw* * *scale*) fits entirely within the destination. See Figure 5 for an example.

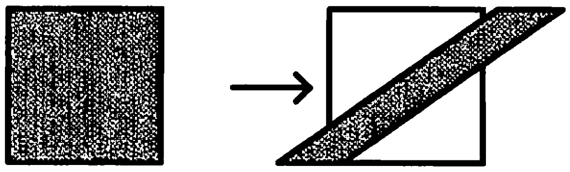


Figure 5: small-dest: only a small part of the destination is derived

9.3 if either succeeds

If either condition "small-source" or "small-dest" holds, then the affine can be implemented as follows (see Figure 6):

1) Copy the needed portion of the source, left-aligned, into the destination.
2) Perform the scale component of the transformation using the Helper subroutine. Perform this right-to-left if *scale* > 1, and left-to-right otherwise.
3) Recursively perform an affine transformation, using only the other components of the original affine transformation.

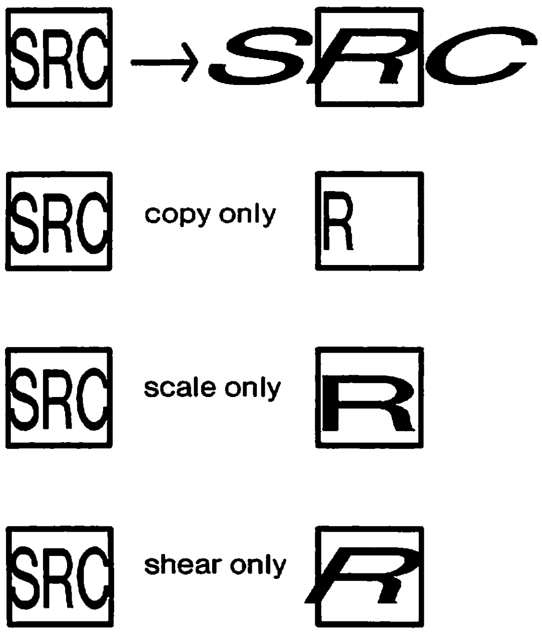


Figure 6: processing small-source

10. The third level of case analysis

Neither of the above special cases may hold. In that case, further analysis is required.

Consider a given (x,y) pixel in the destination. There are three cases for that pixel: it may be far to the left of its source pixel pre-image, it may be far to the right of it, or it may be neither. It is "far to the left" if

$$x' <= src(x',y) - f$$

and "far to the right" if

$$x' >= src(x',y) + f$$

Define the boolean predicate *left*(x',y) to be true if and only if a destination pixel is far to the left, and the boolean predicate *right*(x',y) to be true if and only if a destination pixel is far to the right. Since the transformation is in standard form, *left*(x',y) implies *left*(x'-1,y) and *right*(x',y) implies *right*(x'+1,y).

Due to the tilt in the shear, the rightmost *left*() pixel and the leftmost *right()* pixel may be at a different place on each line. Therefore, we define the function *LEFT*() as the greatest x' such that *left*(w',y) for all w' <= x', for all 0 <= y <= *dw*. *RIGHT*() is similarly defined as the least x' such that all pixels to the right of it have the *right*() property. Figure 8 shows a sample set of *LEFT*() and *RIGHT*() regions.

Intuitively, the *left*() and *right*() predicates detect those destination pixels which are "thrown clear" of their source pixels. The *LEFT*() and *RIGHT*() regions are the largest rectangles contained within the parallelogram-shaped *left*() and *right*() regions.

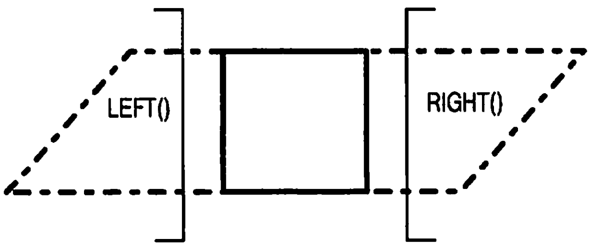


Figure 8: The LEFT() and RIGHT() regions

If *LEFT*() or *RIGHT*() regions exist, then the Helper algorithm is performed on the destination/source regions defined by them, and the problem is trimmed accordingly. However, not all source pixels are freed by this operation: without further help, this operation would quickly grind to a halt.

Therefore, the remaining source image is now split in two along y, and those affine transformations are now recursively analyzed. Splitting the image in this manner reduces the effect of the tilt in the shear, and may create *LEFT*() and *RIGHT*() regions in the sub-images which could not be formed in the originals. In Figure 9, for example, there are no *LEFT*() or *RIGHT*() regions originally, but subdivision creates 4 such regions.

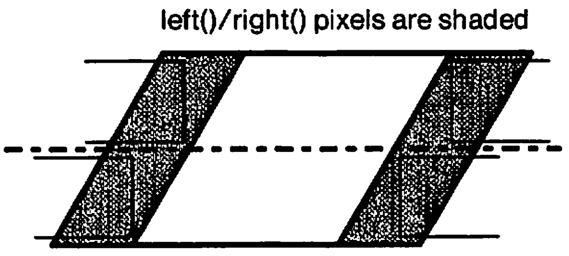


Figure 9: Creating LEFT()/RIGHT() by subdivision.

11. The fourth level of case analysis

It is possible that the shear fits into none of the above categories. This happens when the source image is being very slightly scaled-up and there is little or no shift.

In this case, we have no recourse but to buffer the affine transformation with saves/restores of selected areas of image memory, "stitching" the borders between the panels. Specifically, we now need two buffers B1 and B2, of *f* pixels each. On each scanline:

1) Find the fixed point F, the pixel such that *src*(F,y) = F.

2) Read the pixels from [F-f..F] into B1. These pixels will be written when the left half of the scanline is processed,

but their original values are still needed to process the right half.

3) call the Helper algorithm to evaluate the scanline from [0..F], left-to-right.

4) Read the pixels from [F-f..F] into B2, and then write them with B1.

5) call the Helper algorithm to evaluate the scanline from F onwards, right-to-left.

6) Write the pixels from[F-f..F] from B2.

This algorithm can be performed in *all* cases, but it is very slow: it calls the Helper algorithm twice per scanline, and also must perform 4 buffer I/O operations. If, however, the algorithm is to replace the Catmull-Smith code, then these objections no longer hold, and the algorithm above yields a slower but more general replacement.

12. Summary

An algorithm has been presented to perform in-place affine transformations in constant space. The algorithm subdivides the transformation into a series of smaller transformations. Each smaller transformation is then performed using the Catmull-Smith algorithm. In this way, the new algorithm provides an additional level of capability to a graphics software library, which is particularly appropriate in environments where image sizes are huge and/or processor memory size is limited.

The algorithm works by case analysis, chipping away at the problem by gradually imposing slower and more general algorithms on more difficult portions of the affine transformation: Figure 10 provides a summary.

```
   A summary of the algorithm:
/* split the source into between
 * one and three sub-regions.
 * calls to 'Helper' are of the form
 * Helper(source,dest,
 *       eval order, special notes);
 */

 dir2 = (offset <= 0.0) ?LtoR:RtoL;
 foreach subregion S do
   switch (type) {
   case LEFT:
     /* section 8.1 */
     Helper(S,S,LtoR);
     break;
   case RIGHT:
     /* section 8.2 */
     Helper(S,S,RtoL);
     break;
   case NEITHER:
     /* section 8.3 */
     Helper(s,s,LtoR,scale);
     Helper(s,s,dir2, non-scale);
     break;
   case BOTH:
     if small-source or small-dest {
       /* see figures 4,5, and 6,
          sections 9.1 - 9.3 */
       Copy(needed-part(S),
         left-part(S),left-aligned);
       dir1 = (scale <= 1.0)?LtoR:RtoL;
       Helper(left-part(S),S,
         dir1,scale);
       Helper(S,S,dir2,non-scale);
     } else {
       /* section 10, figure 8 */
       if LEFT region exists {
         Helper(S,LEFT,LtoR);
         S = pruned-part(S);
       }
       if RIGHT region exists {
         Helper(S,RIGHT,LtoR);
         S = pruned-part(S);
       }
       /* figure 9 */
       if (height(S) > 1) {
         Recurse(top-half(S));
         Recurse(bottom-half(S));
       } else {
         /* section 11 */
         stitch
       }
     }
     break;
 }
```

Figure 10: Summary of the Algorithm

13. Extensions and modifications

The algorithm may be extended to handle cases when the source image is a proper subset of the destination image. That case has not been discussed here for presentation purposes. For a discussion of that extension and other implementation issues, the interested reader is referred to [Fishkin89].

14. Acknowledgements

The author wishes to thank Don Conway and David De Francesco (Pixar) for their help with the screen shots, and Andrew Glassner (Xerox PARC) and Tony DeRose (the University of Washington) for their many helpful comments on the organization of the paper.

15. References

[Apple85] Apple Computer, "Inside Macintosh", Addison-Wesley, volume 2, 1985.

[Catmull80] Catmull, Edwin, and Smith, Alvy Ray, "3-D Transformations of Images in Scanline Order", *Computer Graphics* **14**(3), July 1980, pp. 279-285.

[Fishkin89] Fishkin, Ken, "Performing piece-wise in-place affine transformations", Pixar Technical Memo #188, February 1989.

[Fraser85] Fraser, Donald, Schowengerdt, Robert A., and Briggs, Ian, "Rectification of Multichannel Images in Mass Storage Using Image Transposition", *Computer Vision, Graphics, and Image Processing,* **29**(1), pp. 23-26, January 1985.

[Levinthal84] Levinthal, Adam, and Porter, Thomas, "Chap - a SIMD Graphics Processor", *Computer Graphics* **18**(3), July 1984, pp. 77-82.

[Microsoft87] Microsoft Corporation, "Microsoft C Optimizing Compiler: User's Guide", 1987, Chapter 6.

[Porter84] Porter, Thomas, and Duff, Tom, "Compositing Digital Images", *Computer Graphics* **18**(3), July 1984, pp. 253-259.

[Smith87] Smith, Alvy Ray, "Planar 2-Pass Texture Mapping and Warping", *Computer Graphics* **21**(4), July 1987, pp. 263-272.

Appendix: An Example

These pictures show an in-place affine transformation of a 1024 by 768 image, when no more than 256 pixels may be read or written at any time. The image is rotated by 10 degrees and also scaled up by 10%. The dark lines delimit the LEFT, BOTH, and RIGHT regions. The light lines delimit the individual regions passed to Catmull-Smith.

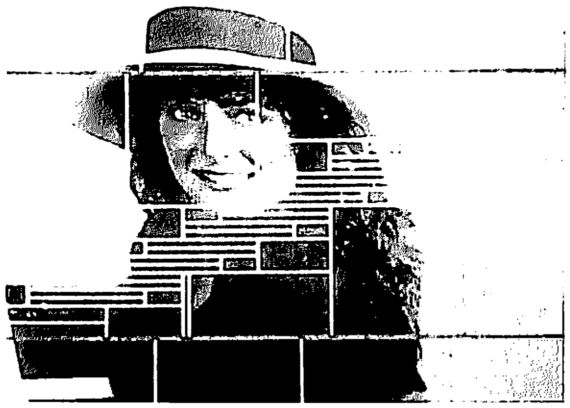


Figure A.3: The First Pass

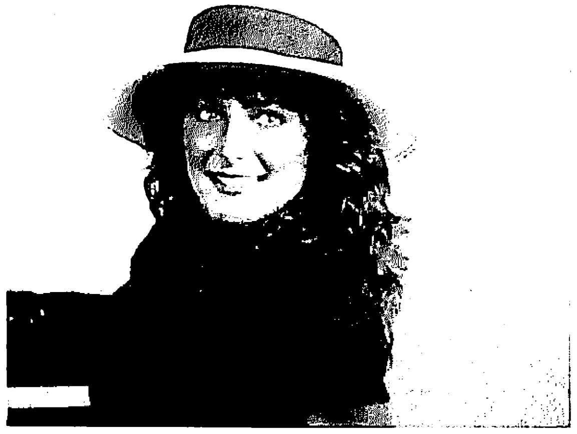


Figure A.1: The Original Image

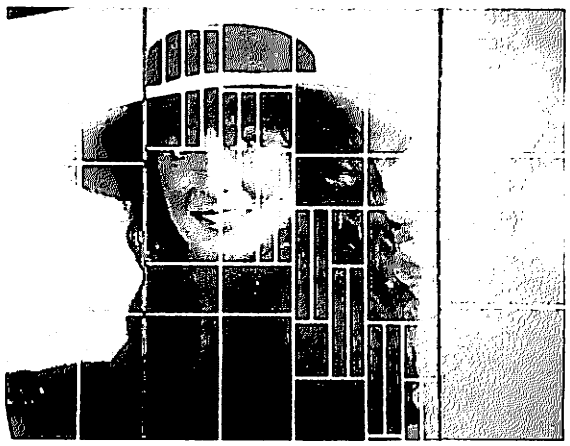


Figure A.4: The Second Pass

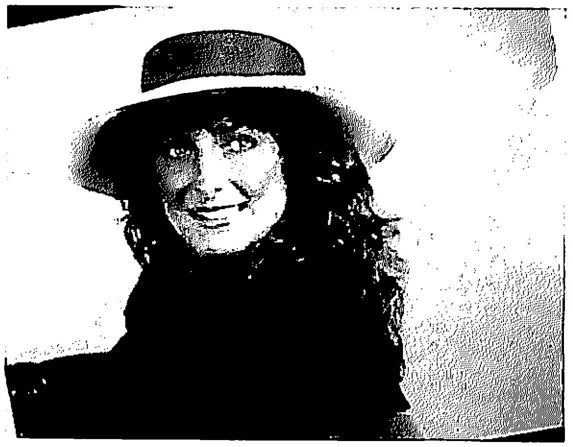


Figure A.5: The Final Image

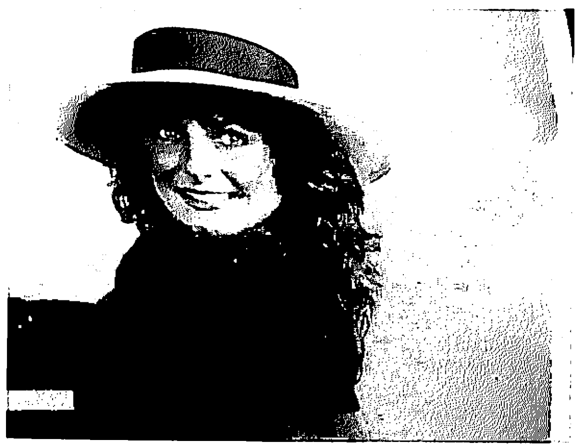


Figure A.2: In the middle of the first pass. The algorithm is transforming the area around the mouth.

Working Together, Virtually

Jin Li and Marilyn Mantei[1]

Department of Computer Science
University of Toronto
Toronto, Ontario M5S 1A4
Tel: (416) 978-6619 Fax: (416) 978-5184
jinli@dgp.utoronto.ca

Abstract

Although technologies such as media spaces have been designed to facilitate collaborative work at a distance, the existing systems have primarily focused on the use of computer managed audio and video as mechanisms to support meetings and video phone calls. Research has shown that frequent and spontaneous informal communication is crucial for project coordination and work progress. It has also found that the amount of collaboration that occurs varies directly with the proximity of co-workers. However, proximity is not always possible or desirable in today's work world. In this paper, we introduce the concept of a virtual open office, a simulated shared open office environment which creates proximity without its inherent disadvantages. We suggest that a large amount of communication among co-workers is not from actual intentional communication contact but from opportunistic contact and environmental scanning in which each individual is picking up valuable coordination information. We propose that it is this aspect of the constant contact of an open office environment which provides the closeness and cohesion necessary for effective work coordination. Based on this premise, we argue for a set of unique user requirements for the virtual open office and demonstrate an instantiation of these requirements in a working prototype, called VOODOO.

Keywords: Groupware, computer supported cooperative work, desktop videoconferencing, informal interaction, virtual spaces.

Introduction

Collaboration occurs frequently in both academic and business environments. Furthermore, it is often impossible to have all collaborators working at the same physical location. This suggests that we need a way to support effective collaboration at a distance. New computer and communication technology make it no longer necessary to assemble all collaborators at the same place, but studies have shown that physical proximity is important for informal communication[10]. It has also been found that better coordination and facilitation are supported by informal communication[5] and that fifty percent of informal work communication is opportunistic[7], e.g., triggered by the sight of another person. Research has also demonstrated the importance of being able to visually identify an opportunity for communication[9].

A large number of media space projects[2, 4, 6, 17] have focused on establishing computer managed audio and video connections to enhance collaboration at a distance. Part of their purpose is to bring back the informal communication that has been lost, but a major portion is also to emulate the rich communication environment of face-to-face contact. It has not always been stated explicitly that the audio and video connections imply a meeting, either prearranged or serendipitous, but the designs of the systems and the discussion of their usage imply that the underlying technology structure is primarily for establishing an intentional contact, what we define in this paper as a meeting. We are taking a different approach in our research. The environment we want to create with the multi-media tools is not one of supporting meetings but one of supporting constant and continuous contact among co-workers. We want to simulate a shared office where the dwellers of the shared office space are miles apart or a simple corridor away. We do not suggest that meeting support by media spaces is inappropriate, but rather, that it is insufficient for the type and amount of communication needed in complex detailed work assignments.

Even when co-workers are not conversing in a shared work environment, they are constantly transmitting details about the joint work. Proximal co-workers, for example, can overhear relevant conversations, view levels of partner progress, perceive changes in project direction, note co-worker's skill advantages and disadvantages, etc. on a real time basis. Furthermore, they can instantly corroborate the acceptableness of any path changes they might make in the joint work. Unfortunately, the large number of advantages of a shared office which promotes this information exchange have been overshadowed by the large number of disadvantages of placing employees in the same room or a weakly partitioned room. We postulate that

[1] Also with the Faculty of Library and Information Science

if media spaces are configured correctly, we can gain back the advantages of the shared office without the incipient disadvantages, e.g., noise and interruptions. Furthermore, we propose that creating such a virtually shared office does not provide a primary benefit of travel cost reduction, but one of closeness and cohesion of co-workers engaged in joint work. Thus, media spaces are not just for enhancing communication at a distance but also for supporting communication within the same building and even on the same floor.

Several existing media space systems incorporate some aspect of supporting informal communication within their structure. Bellcore has built Cruiser[17], a prototype desktop browsing tool which enables unplanned, informal social interaction via audio and video links between co-workers. Other example media space interfaces that have some elements of the virtual office are Polyscope and Vrooms[2] at EuroPARC. Polyscope is a system which distributes digitized images of workers within a building to provide awareness of the other person's presence. It also acts as an interface to the audio and video network so that co-workers can make actual full motion audio and video connections. Vrooms is a modified Polyscope system that addresses some of the social and interface issues found in Polyscope. It employs a stronger spatial metaphor so that people can establish or terminate conversations by entering or leaving a virtual room.

Because it has been found that visual accessibility can be intrusive at times, designers of Cruiser and Vrooms have implemented controls on excessive visual accessibility using rudimentary techniques such as bars (what they call video blinds) crossing the video images or still shots of the co-workers taken at 15 minutes intervals. For Vrooms, Borning and Travers[2] used small video images of co-workers to limit the intrusion of a constantly open video channel. At EuroPARC, the RAVE[6] media space uses user-tailorable buttons to make a variety of audio and video connections to co-workers. One of these buttons is an "office share" button which puts two co-workers in constant continuous full motion video contact similar to the virtual office space we are proposing.

Although the above systems partially support informal interactions at a distance and awareness of co-workers, the approaches taken are ones of simply maintaining the video contact of the media space for a long period of time or of creating a general virtual meeting area where serendipitous contact can occur. This paper extends the continuous contact concept and discusses what other system operations need to be in place to effectively support continuous contact. We introduce the concept of a *virtual open office* — an open office in which physically separated co-workers are in constant contact through open communication channels. We believe that such a virtual open office, although not suitable for all forms of office work, will be useful for detailed technical collaboration, e.g., joint programming. While a media space system is an infrastructure for facilitating collaborative work, a virtual open office is a software environment that is configured within the media space system to satisfy its unique set of user requirements. In the following section, we focus on the user requirements that are appropriate for the virtual open office environment. We then describe our instantiation of these requirements in a system we call VOODOO.

User Requirements

At the University of Toronto, we have built a media space called CAVECAT (Computer Audio Video Enhanced Collaboration And Telepresence)[12]. Whereas CAVECAT is designed for making and breaking video and audio connections with one or more people, the virtual open office is set up to maintain connections with one or more people continuously throughout the workday. This means that co-workers residing in a virtual open office must be accessible for standard CAVECAT video calls just as they would be open to people walking into their shared office. It also means that such calls are made to all members of the open office not just to a single co-worker. Of course, private conversations can ensue just as they might in an open office, but all co-workers would be aware that such private conversations were taking place. Thus by creating a virtual open office in a media space, we add all nature of additional constraints on how that space is to be managed. We have combined existing research on open office communication behavior with experimental observations in our laboratory to generate a list of user requirements for the virtual open office. Although this list is not exhaustive, we have attempted to specify that set of requirements which preserves the advantages of an open office and eliminates its disadvantages. Table 1 lists the entire set of user requirements.

	User Requirements
1	Ability to implicitly establish a co-worker's level of accessibility
2	Ability to enforce reciprocity in information exchange
3	Ability to explicitly set one's level of accessibility
4	Ability to change one's position with respect to co-workers
5	Ability to trivially make verbal and visual contact
6	Ability to trivially close verbal and visual contact
7	Ability to have multi-way conversations
8	Ability to support multi-media information exchange
9	Ability to filter out unwanted noise
10	Ability to discriminate among sounds in the virtual open office
11	Ability to obtain feedback on the communication environment

Table 1: Virtual open office user requirements

Accessibility

In a normal open office environment, co-workers are in constant contact, and thus, they are always available for interaction. In an open office, it is not availability, but co-workers' *accessibility* that is important, e.g., whether the co-worker is at an interruption point in their work or conversation.

Requirement 1: Ability to implicitly establish a co-worker's level of accessibility
Kraut, Egido and Galegher[10] and Allen[1] have postulated that physical proximity is crucial for informal interaction. Our survey of users of the CAVECAT media space has found that users do not make video connections for fear of intruding on the other party[4]. Root[17] has pointed out that people use implicit interaction protocols to indicate a willingness to receive a communication contact. After noticing that someone is in their office, people use cues such as the type of work a potential contact is engaged in to ascertain the occupant's accessibility for interaction. For example, in a conventional open office, co-workers implicitly know not to interrupt a person talking on the telephone. Material placed in one's own workspace as opposed to a more common area such as a book shelf implicitly determines its viewability and thus, accessibility to others. The virtual open office environment should provide similar mechanisms for users to determine the accessibility of co-workers. Because this accessibility is established by spatial arrangements and events that are the normal course of work, this same implicitness needs to be available in the virtual system.

Requirement 2: Ability to enforce reciprocity in information exchange
When people used EuroPARC's media space, Polyscope, video symmetry was almost never requested, that is, users did not ask to see who was looking at them. Users may not be aware of the unequal information exchanges supported by the system. In an open office, viewing is reciprocal. If I can see someone I know that person can see me. In a media space, this is not necessarily true. Furthermore, although I can see someone, they might be able to see more of me and at a much finer level of detail. This unequal exchange can cause severe imbalances in relations and exchanges. It is therefore necessary to explicitly enforce reciprocity in all information sharings in a virtual open office if natural coordination relations are to be maintained.

Requirement 3: Ability to explicitly set one's level of accessibility
Chatting with co-workers is often a hindrance (albeit enjoyable) to work in an open office. So is maintaining document privacy. Therefore, control over conversational and workspace accessibility is an essential need for the virtual open office.

Requirement 4: Ability to change one's position with respect to co-workers
Marmolin, Ahlstrom and Ropa[14] have found that people use a large video image for discussion, but when they are working intensely on their own tasks, they use a small video image for checking the communication status of their co-workers. Our own laboratory studies of two individuals working on a joint programming task over the media space showed that people did not use the visual image of their partner when they were focussing on the task. They only glanced at the video occasionally and sat away from the video screen. When they were negotiating a detail about the task, they moved their chairs directly in front of the video image and engaged in a more direct face-to-screen contact. In an open office, people often learn a large amount about their co-workers by glancing around their co-workers' office or looking at what's on their co-workers' desk or book shelf. By walking over to the other's desk, people are able to get a closer look. A virtual open office environment should be capable of handling the close contact as well as the environmental scanning which people use in their daily work to gain information about their colleagues.

Communication Cost
There are financial and behavioral costs associated with establishing communication. The behavioral cost aspect is more important for informal interaction at a distance. Different communication media have been shown to affect the collaboration process. The more limited the communication medium is, the less effective the collaboration process. If the behavioral cost is high, such as remembering and pushing several digits on a telephone, waiting for an answer and establishing a communication, a mental cost/benefit tradeoff will be calculated and communication below a particular threshold will not take place — even if the communication would have transmitted important information. Allen[1] has shown exponential drop-offs in the frequency of communication between co-workers as the physical distance increases. In their study of communication, Kraut et al.[11] found that 52% of the conversations involved people located off the same corridor and 87% of the conversations took place among people who shared the same floor of a building. We believe that it is the cost of making informal contacts when distance increases that causes their significant fall off.

Requirement 5: Ability to trivially make verbal and visual contact
In the absence of close proximity, we need to make the initiation cost of communication very low to encourage frequent and spontaneous communication. In the joint programming studies we conducted in our simulated virtual open office environment, we found that conversations opened and closed without any formal protocols just as they might in regular exchanges. We noted that no conversation commencement and termination protocols occurred even when there existed gaps of five to ten minutes between verbalizations. We assume a state of communication that no longer needs formal contact protocols, much like that of a continuous conversation with pauses. Contact should be as easy as starting to talk or raising one's head to gain other's attention.

Requirement 6: Ability to trivially close verbal and visual contact
It is equally important to have a low behavioral cost for terminating communication. Co-workers should not need to follow a formal closing protocol. After all, the conversation is assumed to be continuous in this state, only punctuated by pauses. Our studies of joint programming in our simulated virtual open office support this behavior as well.

Requirement 7: Ability to have multi-way conversations
It should be easy to have a third party join or leave a conversation. People should be able to make entrances

into a virtual open office conversation similar to the way they normally would in a physical open office. However, entrants into a physical open office often walk up to one person's desk and engage in a private conversation. Such activities should also be supported in the virtual open office including the one in which the two conversants leave the open office for an even more private exchange. As in a regular open office, concurrent conversations of several subgroups of co-workers should be permitted to occur with listening support for other co-workers. However, such listening support should not permit others to overhear private verbal exchanges.

Information Sharing
Information sharing adds semantic content to a conversation and provides an underlying context for the discussion[11]. Tang[19] has demonstrated that shared drawing space is not only useful for storing information and conveying ideas, but also for developing ideas and mediating interactions. Ishii[8] has shown the problems of work integration in shared computer environments and presented a video solution for seamless shared workspaces. Lauwers and Lantz[13] have suggested a set of user requirements for shared window systems to support sharing using existing collaboration transparent applications. Although document sharing is important, so is document privacy. Oldham and Rotchford[15] have shown that in an open office, co-workers preserve their ownership of a workspace by placing personal spatial markers around that space. There is the need for both defining and preserving the ownership of people's workspace when that space becomes virtual. Thus there is a need for both workspace sharing in a virtual open office but also one of allowing a user to establish limits on this sharing.

Requirement 8: Ability to support multi-media information exchange
In our shared programming study, we emulated a virtual open office by setting up two offices back-to-back and passing cables for exchanging computer screen images and camera images to monitors on the other side of the wall. We used Y-cables to split each computer's video output and send it to a second screen on the other side of the wall. Thus, each programmer had a view of what activities and code the other programmer was working on as well as video and audio connections to their co-worker. We found that an extensive amount of time was spent looking at the other person's code and hand copying it from one terminal to another. Each person also pointed to elements in the other person's code but this action was not visible by the second person in our rudimentary setup. From these exploratory studies we ascertained that the virtual open office environment should have all shared work information available to all parties in the office. Co-workers should be able to easily share objects such as drawings and text[3]. Telepointing facilities should be provided for people to easily refer to objects in the shared workspace. The system should be able to handle off-line as well as on-line material and to allow synchronous annotation of screen objects. Co-workers should be able to easily demonstrate processes. For example, a person may show the execution of a program which contains bugs and ask the other person how to correct the program. Co-workers should also be able to easily copy relevant screen objects from their co-workers' machine.

Environmental Improvement
Use of open audio and video channels leads to concerns about preserving an individual's privacy[2]. An open office is not private[16, 18], but workers are always aware of this lack of privacy. A virtual open office is problematic because the cues to indicate a privacy problem may no longer be available. Privacy issues in a virtual open office are very different from that in a media space system. A virtual open office does not provide co-workers with total privacy, thus it is not a suitable environment for people who want to work privately. Co-workers in a virtual open office face the problem of distractions such as noise. Noise has been a common complaint of workers in open offices[18]. Ambient noise in the office and noise generated from co-workers' chatting and typing forces people to make an extra effort to concentrate on their work. One of the problems with an open office is that there is so much noise that it is very difficult to discriminate between the sounds we want to hear and those we want to shut out. The inability for participants of a media space to localize sound makes it more problematic because the location cues used for sound filtering are lost. During CAVECAT sessions, on many occasions several people would answer their phone when the phone rang in another office. Users of our CAVECAT system have expressed the desire for system generated cues to help them spatially separate sound sources[4].

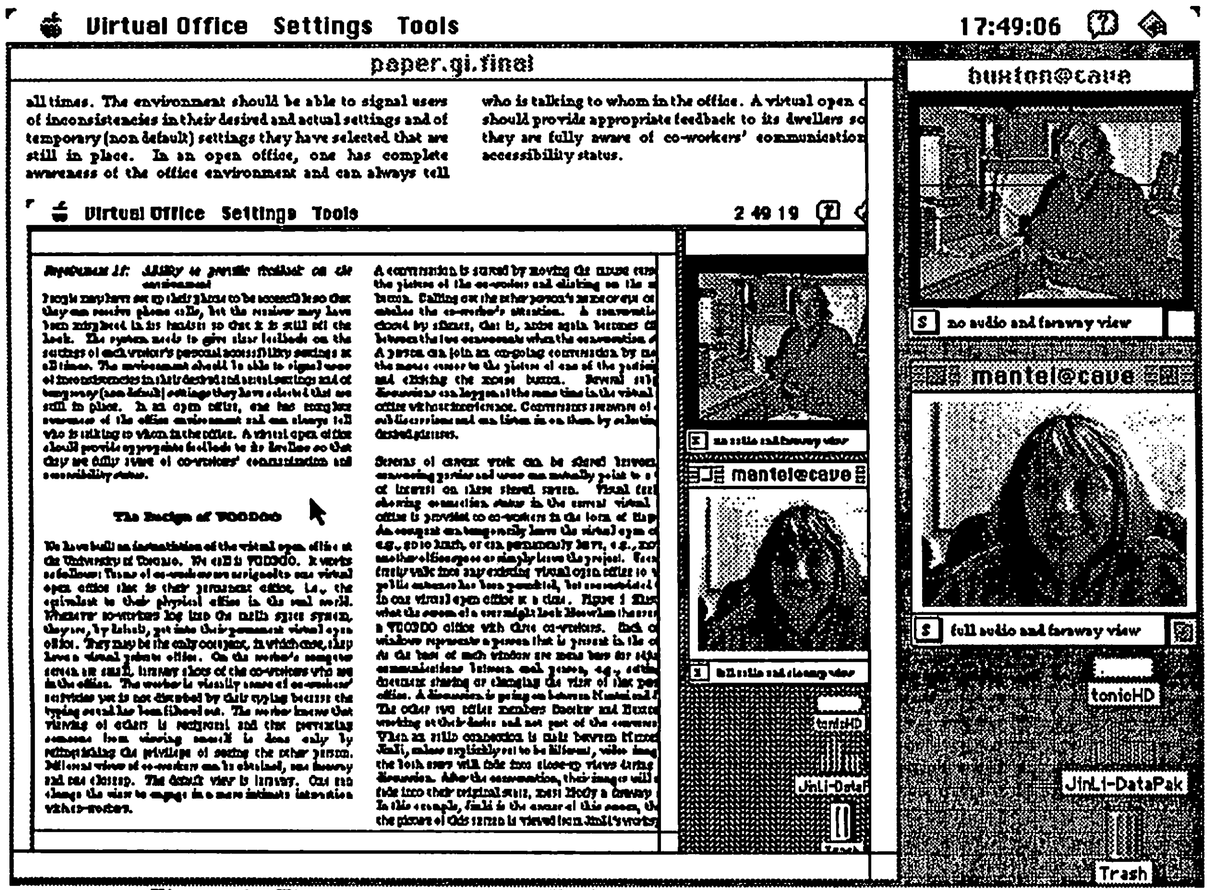
Requirement 9: Ability to filter out unwanted noise
Limited options exist when a co-worker is very noisy in a physical open office but the virtual open office makes it simple to quiet a noisy inhabitant just by lowering a volume control. Thus, one should be able to muffle out the noise generated by a co-worker. This silencing capability needs to be reciprocal, i.e., users should be able to prevent a conversation from reaching others as well as inhibiting conversations from disturbing them.

Requirement 10: Ability to discriminate among sounds in the virtual open office
Through the use of technology, e.g., a three dimensional sound system or different phone rings for different people, the system should provide implicit cues to people for spatial separation of sound sources and identification of salient signals.

System Status
Awareness of both the physical and social environment is required for maintaining informal activities in a virtual open office. Information and feedback should be provided so that collaborators have a constant overall picture of the work environment.

Requirement 11: Ability to provide feedback on the environment
People may have set up their phone to be accessible so that they can receive phone calls, but the receiver may have been misplaced in its handset so that it is still off the hook. The system needs to give clear feedback on the settings of each worker's personal accessibility settings at

all times. The environment should be able to signal users of inconsistencies in their desired and actual settings and of temporary (non default) settings they have selected that are still in place. In an open office, one has complete awareness of the office environment and can always tell who is talking to whom in the office. A virtual open office should provide appropriate feedback to its dwellers so that they are fully aware of co-workers' communication and accessibility status.



all times. The environment should be able to signal users of inconsistencies in their desired and actual settings and of temporary (non default) settings they have selected that are still in place. In an open office, one has complete awareness of the office environment and can always tell who is talking to whom in the office. A virtual open office should provide appropriate feedback to its dwellers so that they are fully aware of co-workers' communication and accessibility status.

Figure 1: Three co-workers use VOODOO to work in a virtual open office

The Design of VOODOO

We have built an instantiation of the virtual open office at the University of Toronto. We call it VOODOO. It works as follows: Teams of co-workers are assigned to one virtual open office that is their permanent office, i.e., the equivalent to their physical office in the real world. Whenever co-workers log into the media space system, they are, by default, put into their permanent virtual open office. They may be the only occupant, in which case, they have a virtual private office. On the worker's computer screen are small, faraway shots of the co-workers who are in the office. The worker is visually aware of co-workers' activities yet is not disturbed by their typing because the typing sound has been filtered out. The worker knows that viewing of others is reciprocal and that preventing someone from viewing oneself is done only by relinquishing the privilege of seeing the other person. Different views of co-workers can be obtained, one faraway and one closeup. The default view is faraway. One can change the view to engage in a more intimate interaction with co-workers.

A conversation is started by moving the mouse cursor to the picture of the co-worker and clicking on the mouse button. Eye contact or verbal hailing catches the co-worker's attention. A conversation is closed by a mouse click to toggle to an audio off state. A person can join an on-going conversation by moving the mouse cursor to the picture of one of the participants and clicking the mouse button. Several subgroup discussions can happen at the same time in the virtual open office without interference.

Screens of current work can be shared between the conversing parties and users can mutually point to a topic of interest on these shared screens. An occupant can temporarily leave the virtual open office, e.g., go to lunch, or can permanently leave, e.g., move to another office space or simply leave the project. Users can freely walk into any existing virtual open office to which public entrance has been permitted, but are restricted to be in one virtual open office at a time. Figure 1 illustrates what the screen of a user might look like when the user is in a virtual open office called *cave* with two co-workers. Each of the windows represents a person that is present in the office. At the base of each window are a button labelled *s* for adjusting communication states between each person, e.g., changing the view of that person's office, and a panel for displaying the current communication state. A discussion is going on between Mantei and JinLi. The other office member Buxton is working at his desk and not part of the conversation. When an audio connection is made between Mantei and JinLi, unless explicitly set to be different, video images of both users will fade into close-up views during their discussion. The white border around the image indicates the audio is on, while the black border indicates audio is off. After the conversation, their images will again fade into their original state, most likely a faraway shot. In this example, JinLi is the owner of this screen, that is, the picture of this screen is viewed from JinLi's workspace.

	User Requirement	Design Solution
1	Ability to implicitly establish a co-worker's level of accessibility	Constant open video communication channel
2	Ability to enforce reciprocity in information exchange	Enforced audio, video and computer screen symmetry
3	Ability to explicitly set one's level of accessibility	Explicit software settings available
4	Ability to change one's position with respect to co-workers	Close-up and fly on the wall cameras
5	Ability to trivially make verbal and visual contact	Open video channel and mouse click on image
6	Ability to trivially make verbal and visual contact	Open video channel and mouse click on image
7	Ability to have multi-way and concurrent conversations	Software to explicitly select participants for a conversation
8	Ability to support high quality information exchange	Shared computer screens, telepointing, workspace viewing and (not implemented)document tray metaphor
9	Ability to filter out unwanted noise	Muffler metaphor(partially implemented)
10	Ability to discriminate among sounds in the virtual open office	3D sound system with MIDI(not implemented)
11	Ability to provide feedback on the environment	Diagrams of connections(not implemented)

Table 2: Design solutions to the user requirements

VOODOO is implemented on a Macintosh computer using a client-server architecture. The server resides on a SPARCStation and keeps an updated database of resources, connections and activities in the virtual open office. Table 2 illustrates the proposed solutions to the user requirements. We discuss the features of VOODOO in the following sections.

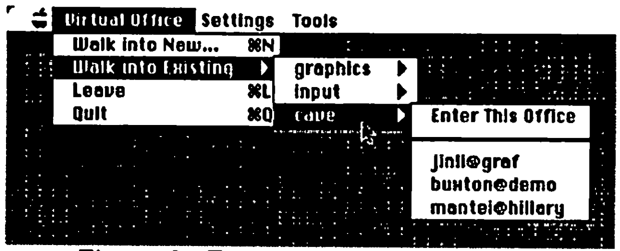


Figure 2: Enter a virtual open office

Figure 2 shows how a user enters a virtual open office by selecting the appropriate office from the virtual offices menu displayed at the top of the Macintosh screen. The menu dynamically displays all the existing virtual offices that are accessible to users and shows who are currently working and where they are physically located in the respective virtual offices. Once a user has entered a virtual office, all video images of co-workers in the office appear as small windows on the user's screen. Via the open video channel, the user can easily tell who is currently working in the office and implicitly establish co-workers' accessibility. This satisfies user Requirement 1. VOODOO uses full motion but low spatial resolution video for the faraway shot. Full motion video provides users with real time visual awareness of events, while low quality video preserves co-workers' privacy and prevents excessive visual intrusion. When co-workers are engaged in a conversation to negotiate details about a task, full motion closeup video is used to enhance information exchanges. Since co-workers can see each other when they are visually accessible, this meets the reciprocity constraint of Requirement 2. Associated with each individual image on the screen is a set of communication and workspace attributes that specify a person's accessibility. They can be explicitly set. For example, a user can turn off the audio coming from a colleague who is unusually noisy. This capability satisfies Requirement 3.

VOODOO uses several video cameras to provide users with different views of their co-workers so that the complete office scene of a co-worker can be viewed and a richer information exchange can occur. The closeup camera is placed on top of the computer screen and captures a head and shoulders image of a co-worker. The "fly on the wall" camera is mounted on the office wall and captures the interior of the office. Figure 3 shows how a user can move closer to another co-worker for an intimate interaction or further away to get a broader view of the office by holding down the mouse button on a video image to get a popup list. This setup fulfills Requirement 4.

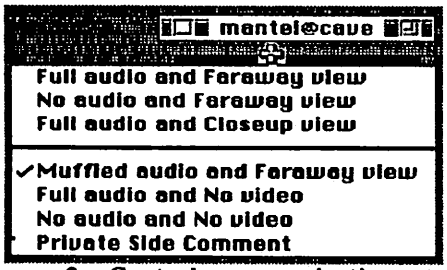

mantel@cave
Full audio and Faraway view
No audio and Faraway view
Full audio and Closeup view
✓Muffled audio and Faraway view
Full audio and No video
No audio and No video
Private Side Comment

Figure 3: Control communication states

To initiate verbal contact, users need to use the mouse to click on the video image of the co-worker they wish to communicate with. Although the action must be done explicitly, it has the explicitness and ease of use that are intrinsic to the social protocol of calling out a co-worker's name in a face-to-face contact. Another approach that is being considered is the usage of sensors to detect head movement. As a user raises his or her head and starts talking, the head direction selects the contact person and a voice activated audio connection is made. Closing a conversation is also done by a mouse click. These interface features meet the requirements of low behavioral cost for making and breaking communication (Requirements 5 and 6). The current VOODOO system does not have the voice activated audio feature, but we are working to set up the appropriate hardware and software to incorporate it in the next version.

Users can selectively decide who is not supposed to overhear the current conversation with menu selections. Alternatively, a user just needs to mouse click on the picture of one of the participants of a conversation to smoothly join the meeting. This interface design meets Requirement 7.

We propose a document tray metaphor for users to share documents while preserving ownership of workspace. When a user places a document in a co-worker's document tray, that co-worker immediately gains viewing access to the document. However, should the user decide that the co-worker no longer has access to the document, the user can remove the document from the co-worker's tray. Even if the co-worker is reading the document, when the user takes the document, the co-worker loses access to the document. A telepointer is available in a unique color for each user. This allows each co-worker sharing a document to point to important information they are discussing. Their telepointers can be seen on all other screens displaying the document. Documents can be all possible documents generated by Macintosh applications. We have provided users with an application tools chest (Figure 4). Users can run other collaborative or single-user Macintosh applications within the VOODOO system to perform different tasks with their co-workers. Through the use of the document tray metaphor, telepointers and shared computer screens, a high bandwidth of information can be easily exchanged and thus, Requirement 8 is fulfilled. The document tray metaphor has not been incorporated into our initial version of VOODOO.

The muffler metaphor is used to circumvent the noise problem in an office. Users have the option to muffle the audio signals at several different levels. They can muffle noise such as a co-worker's typing sound but permit normal human voice to be transmitted across the audio connection. The level of muffling is controlled by a slider interface. A user sets the level by adjusting the slider bar. When a user is disturbed by the typing sound of the co-workers, after adjusting the muffling level, the typing sound will fade to the background and be filtered out completely. This feature, which satisfies Requirement 9, has not been implemented in the current system prototype.

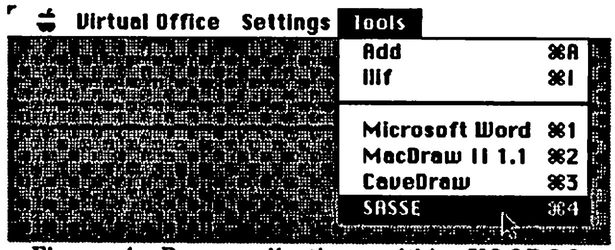


Figure 4: Run applications within VOODOO

One of the problems with current media space is the lack of proper audio feedback for users to spatially separate their co-workers. Three dimensional sound hardware and software systems can reproduce audio while preserving its spatial properties. This permits a user to recognize where the sound is coming from. If such a system were implemented in our interface, Requirement 10 could be met.

A diagram of current communication connections is provided to co-workers to give feedback on who is talking to whom in the virtual open office so that co-workers are constantly aware of communication activity in their environment. For example, when a full audio connection is established between two co-workers, the diagram could show cables connecting one co-worker's microphone connected to the other's speaker. This would then meet Requirement 11.

Conclusion

A unique set of user requirements for the virtual open office concept has been presented. VOODOO, a working prototype of the virtual open office concept, provides an environment for collaborators so that easy and seamless informal interactions can be achieved and so that co-workers are constantly aware of their colleagues' activities without problematic interruptions and noisy environments. Further detailed user testing and field study are required to evaluate the usefulness of the virtual open office.

Acknowledgements

For research support, the authors are indebted to the Natural Sciences and Engineering Research Council of Canada (NSERC), the Information Technology Research Centre of Ontario (ITRC), Apple Computer Inc., Digital Equipment

Corporation, IBM Canada's Laboratory Centre for Advanced Studies in Toronto and Rank Xerox EuroPARC.

References

1. Allen, T. J. (1977) Managing the flow of technology. Cambridge, MA: MIT Press.
2. Borning, A. and Travers, M. (1991) Two approaches to casual interaction over computer and video networks. *Proceedings of CHI'91 Conference on Human Factors in Computing Systems,* New Orleans, LA, April, 1991, 13-19. New York: ACM Press.
3. Buxton, W. A. (1991) Telepresence: Integrating shared task and personal spaces. Submitted to *Groupware'91.*
4. Buxton, W. A. and Sellen, A. J. (1991) Interfaces for multiparty video conferences. In *Proceedings of CHI'92 Conference on Human Factors in Computing Systems.* Monterey, CA, May, 1992.
5. Daft, R. L. and Lengel, R. H. (1986). Organizational information requirements, media richness, and structural design, *Management Science,* vol. 32, 554 - 571.
6. Gaver, W. W., Moran, T., MacLean, A., Lovstrand, L., Dourish, P., Carter, C. A. and Buxton, W. (1991) Working together in media space: CSCW research at EuroPARC. *Proceedings of the Unicom Seminar on computer Supported Cooperative Work: The Multimedia and Networking Paradigm.* London, England, July, 1991.
7. Gullahorn, J. T. (1952). Distance and friendship as factors in the gross interaction matrix, *Sociometry,* vol. 15, 123 - 134.
8. Ishii, H. and Miyake, N. (1991) TeamWorkStation: Towards an open shared workspace. *Communication of the ACM.*
9. Kendon, A. and Ferber, A. (1973). A description of some human greetings. In Michael, R. and Crook, J. (Eds.) Comparative Ecology and Behavior of Primates, 591 - 668. London: Academic Press.
10. Kraut, R., Egido, C. and Galegher, J. (1990). Patterns of contact and communication in scientific research collaboration. In J. Galegher, R. Kraut and C. Egido (Eds.) Intellectual Teamwork: Social and Technological Foundations of Cooperative Work, 327-350. Hillsdale, NJ: Lawrence Erlbaum Associates, Inc.
11. Kraut, R. E., Fish, R. S., Root, R. W., and Chalfonte, B. L. (1990). Informal communication in organizations: Form, function, and technology. In Oskamp, S. and Spacpan, S. (Eds.) People's Reactions to Technology, 145 - 199. Beverly Hills: Sage Publications.
12. Mantei, M.M., Baecker, R.M., Sellen, A.J., Buxton, W.A., Milligan, T. and Wellman, B. (1991) Experience in the user of a media space. *Proceedings of CHI'91 Conference on Human Factors in Computing Systems,* New Orleans, LA, April, 1991, 203-208. New York: ACM Press.
13. Lauwers, J. C. and Lantz, K. A. (1990) Collaboration awareness in support of collaboration transparency: Requirements for the next generation of shared window systems. *Proceedings of CHI'90 Conference on Human Factors in Computing Systems,* Seattle, WA, April, 1990, 303-311. New York: ACM Press.
14. Marmolin, H., Ahlstrom, B. and Ropa, A. (1991) *A laboratory case study of a co-working task using picturephone, shared screens and speech communication.* Unpublished report, Laboratory of Human computer Interaction, Linkoping, Sweden.
15. Oldham, G. and Rotchford, N. (1983) Relationships between office characteristics and employee reactions: a study of the physical environment, *Administrative Science Quarterly,* vol. 28, December, 1983, 542-556.
16. Oldham, G. (1988) Effects of changes in workspace partitions and spatial density on employee reactions: a quasi-experiment, *Journal of Applied Psychology,* vol. 73, May, 1988, 253-258.
17. Root, R.W. (1988). Design of a multi-media vehicle for social browsing. In *Proceedings of CSCW'88 Conference on Computer-Supported Cooperative Work,* Portland, OR, September 1989, 25-38. New York: ACM Press.
18. Sundstrom, E. (1986). *Work Places: The Psychology of the Physical Environment in Offices and Factories.* Cambridge: Cambridge University Press.
19. Tang, J.C. and Minneman, S.L. (1990). VideoDraw: a video interface for collaborative drawing. *Proceedings of CHI '90 Conference on Human Factors in Computing Systems,* Seattle, WA, April, 1990, 313-320. New York: ACM Press.

Telepresence: Integrating Shared Task and Person Spaces

William A. S. Buxton

Computer Systems research Institute
University of Toronto
Toronto, Ontario, Canada M5S 1A4

Abstract

From a technological and human perspective, shared space in remote collaboration has tended to focus on shared space of either the people or the task. The former would be characterized by traditional video/teleconferencing or videophones. The latter could be characterized by synchronous computer conferencing or *groupware*.

The focus of this presentation is the area where these two spaces meet and are integrated into what could be characterized as video-enhanced computer conferencing or computer-enhanced video conferencing.

From the behavioural perspective, the interest lies in how - in collaborative work - we make transitions between these two spaces. For example, in negotiating, the activity is mainly in the shared space of the participants themselves, where we are "reading" each other for information about trust and confidence. On the other hand, in preparing a budget using a shared electronic spreadsheet, for example, the visual channel is dominated by the task space.

How well systems affords natural transitions between these spaces will have a large impact on their usability, usefulness, and acceptance. Consequently, we investigate the design space and some of the issues affecting it.

Keywords: Human-computer interaction, CSCW, Videoconferencing, Groupware.

Introduction

Groups play an important role in our work-a-day life. Physical proximity facilitates interaction among group members. Even splitting groups across two floors of the same building can have a negative effect on group dynamics (Kraut & Egido, 1988), yet in many organizations groups are distributed across campuses, cities, countries, or even the globe. The health of these organizations is tightly coupled to the ability to maintain a sense of "group," despite such distances. Our interest lies in developing *telepresence* technologies appropriate for fostering such maintenance.

As we use the term, telepresence is the use of technology to establish a sense of shared *presence* or shared *space* among geographically separated members of a group. The topic is of particular interest now due to the ongoing convergence and affordability of the requisite computer, telecommunications and audio/video technologies; however, if these technologies are going to be deployed in anything other than a tail-wagging-the-dog technology-driven manner, we must first develop a better understanding of what we mean by "shared space" or "shared presence" in the context of group interactions.

In what follows, we begin to investigate what is shared in various types of group interactions, and some of the technological implications of supporting such sharing. Our purpose is "consciousness raising" rather than the presentation of formal theories or models. Our case is made primarily through the use of examples. Our hope is to provide some foundation for making better design decisions and better exploiting the potential of existing and evolving resources.

Starting from the Known

The terms "meeting" or "group interaction" are almost devoid of information since they encompass such a broad range of activities. Each has its own set of properties and purposes. Only by understanding these properties can we hope to design the appropriate affordances into supporting technologies.

This is nothing new. Take architecture as an example. Because it is a mature discipline, we think of it as part of the general ecology of work, rather than as a technology. Yet a technology it is, and very much a technology to support group activities. Consider, then, the different types of group activities that are a part of our everyday work, and how the affordances of this technology have been designed to support them. We clearly understand differences of purpose, and choose the space (office, lounge, laboratory, board room, gym, lunch room, etc.) accordingly.

Because the technology is mature, we have a good sense of how to match the activity to the space. In order to be considered mature, the electronic meeting spaces of telepresence must meet the same dual criteria of supporting a comparably rich range of group activities and doing so in such a way that users have the same transparent sense of appropriateness of space-to-activity.

To speak of "videoconferencing" or "telepresence" is analogous to speaking about "buildings." While having some value, the grain of analysis is too course to foster an understanding of what goes on "inside." While we would never do so with rooms in a building, our current level of (im)maturity with electronic spaces has a tendency towards "one size fits all." This is something that we must break out of. The range of electronic meeting spaces, like the range of spaces in a well-designed building, must match the richness and range of meeting types. As a start to achieving this, we can move from the level of "buildings" to that of "rooms" and try to gain some insight into the nature of some of the different spaces that we want to share.

Person and Task Spaces

In what follows, we are going to consider presence in terms of two spaces: that of the person and that of the task. From even such a simple cut, several interesting insights emerge.

What we call shared *person space* in telepresence is the collective sense of copresence between/among group participants.[1] This includes things like their facial expressions, voice, gaze and body language.

By shared *task space* we mean a copresence in the domain of the task being undertaken. If we were doing a budget, for example, this might mean that each of us has the budget in front of us in the form of a shared speadsheet. Despite the distance, each of us can act upon it to make changes, annotations, or just to indicate cells that are the subject of discussion.

Sometimes the person and the task spaces are the same. One example would be in negotiations or counseling. Here a major part of the task involves "reading" the other person, such as to evaluate confidence or trust[2]. In other cases, such as our budget example, person and task spaces are more distinct. In what follows we shall see that different technologies lend themselves to differing degrees in supporting these two spaces. The point that we are leading to is that one of the most important attributes of a system is the *seamlessness* of their integration (Ishii & Miyake, 1991), and how well they match the needs of the activity to be supported.

Video Conferencing and Person Space: Some Examples

Traditional videoconferencing is a fairly good example of attempting to establish shared person space. While nobody would ever be fooled into thinking that the remote parties were actually in the same room, one can at least maintain an awareness of who is present and get a general reading of their body language, for example. The absence of checks like, "Are you still there Marilyn?" that are characteristic of telephone conferences is an example of what video contributes to maintaining a sense of personal presence.

[1]This is in contrast to 'personal space" which carries the connotation of privacy, not sharing. Thanks to Hiroshii Ishii for making this point and prompting me to change my terminology.

[2] This is sufficiently important that we might well refer to these as *trustification*, rather than *communication* technologies.

Fig. 1, illustrates one example of how video can be used to maintain a sense of personal presence in a four-way meeting.

Figure 1: *A videoconference involving four participants.*

The quality of the shared person space can be improved through design, however. Below, we give some examples that illustrate the breadth of the available design space. While many of these techniques are well known, few have found their way into mainstream videoconferencing. If establishing a strong sense of person space is important, then perhaps current practice needs to be reexamined.

For example, traditional videoconferencing is typically afflicted by an inability to establish eye contact among participants. This is because of the discrepancy of the position of the image of your eyes on my monitor and the position of your effective (surrogate) eyes, the camera, which is typically located on top of the monitor.

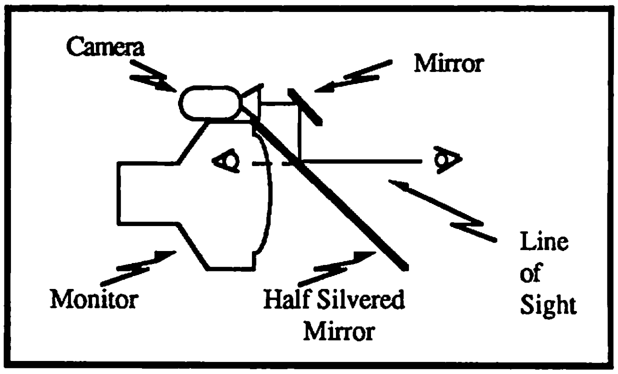


Figure 2: The Reciprocal Video Tunnel. Through the combination of a mirror and half silvered mirror, there appears to be direct eye-to-eye contact. The mirrors effectively place the camera right in the line of sight. A close approximation to reciprocal eye contact can be obtained if both parties are using such an arrangement (from Buxton & Moran, 1990).

By adopting teleprompter technology from the broadcast industry, this problem of eye contact can be largely overcome. The technique is shown in Fig. 2, as it was implemented by William Newman at Rank Xerox EuroPARC. Two mirrors, one of which is half silvered, are used to reflect what is in front of the screen up to the camera, which is mounted on top of the monitor.

The use of such teleprompter-like technology to obtain eye contact is not new. It was patented in 1947 (Rosenthal, 1947), has been studied by Acker & Levitt (1987) and used by Newman (as mentioned above), and more recently in a novel form in the *Clearboard* system (Ishii & Kobayashi, 1992). While it's use is not widespread in videoconferencing, users report greater comfort and naturalness in face-to-face meetings carried out using the technique.

Portrait painting provides the lead for another approach to augmenting the nature of personal presence using video. Video monitors have what is called a landscape *aspect ratio* (the ratio of the width to the height of a video monitor), because of their horizontal orientation. A very simple trick is to turn the camera and monitor at both ends of a conference onto their sides. The result, illustrated graphically in Fig. 3, is a *portrait style* aspect ratio.

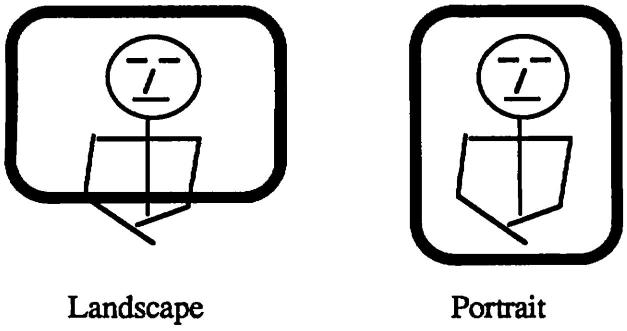


Figure 3: *The effect of switching from Landscape to Portrait aspect rations in person-to-person video conferences. Note that, all other things being equal, in the portrait orientation, the hands and desk-top are visible, thereby adding to the ability to use a richer vocabulary of body language in the dialogue.*

When the image of a single person is to be transmitted, more of that person's body is visible without changing the size or resolution of the face. Consequently, in the example, the hands of the participant are visible in the portrait version, as would be the desk-top. The design affords access to a richer vocabulary of body language. As a prototype unit built by colleagues from the University of Ottawa has shown, this approach can be particularly effective where screen size is constrained, such as with small desk-top units, since a larger screen surface is available for a given width of package.

Next, let us consider the case of where we want to have a meeting involving the participation of more than two sites. At the University of Toronto, we have developed a system called *Hydra,* in which each remote participant is represented by a *video surrogate* (Sellen, Buxton & Arnott, 1992; Buxton & Sellen, 1991)[3]. The technique involves having a separate camera, monitor and speaker for each remote participant. As we have implemented it, these components are housed in very compact desk-top units, as shown in Fig. 4.

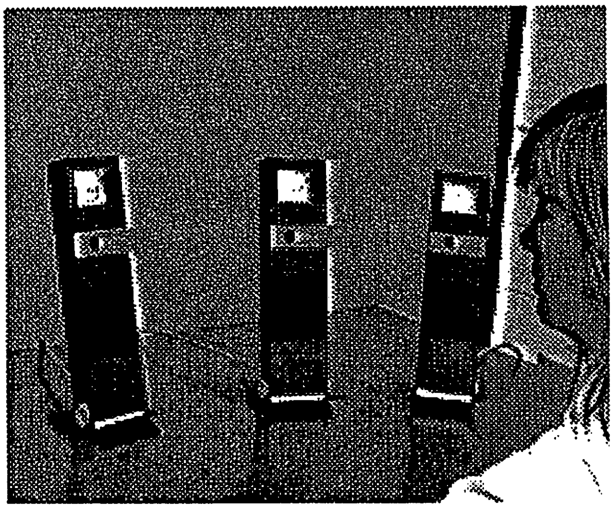


Figure 4: *A user is seated in front of three Hydra units. In the photo, the Hydra units sit on the table in the positions that would otherwise be occupied by three remote participants. Each Hydra unit contains a video monitor, camera, and loudspeaker. A single microphone conveys audio to the remote participants (From Buxton & Sellen, 1991).*

Using this arrangement, the notion of person space is preserved. Because of this it is potentially much easier to maintain awareness of who is visually attending to whom, and to take advantage of conversational acts such as head turning. The idea behind the design is to take advantage of existing skills used in the work-a-day world. For example, in comparing this technique to other approaches to supporting multiparty conferences (Sellen, 1992), the *Hydra* units were unique in their ability to support parallel conversations, which naturally occurred in the face-to-face base-line condition.

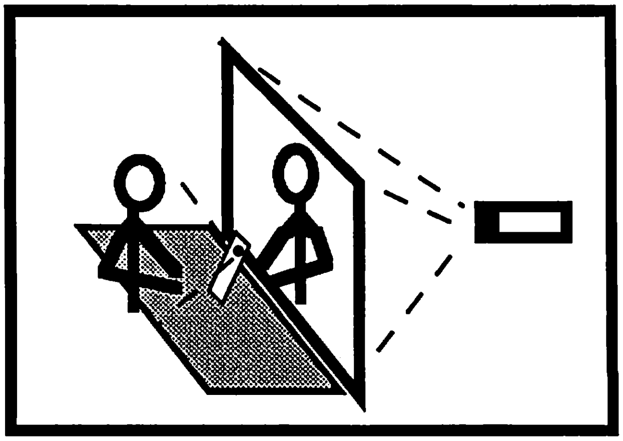


Figure 5: *Using a projected image to obtain a life-sized cross-table presence. Participants are captured using a miniature camera on the desk-top, so as to minimize obstruction of the projectted image. In our installation, we use one of the Hydra units (camera only), illustrated in Fig. 4.*

[3] After the fact, we have become aware that this approach was first developed by Fields (1983).

Finally, the effect of scale has been little explored as a factor that influences a sense of presence. There is a strong possibility that if the video images are life size, that social relationships, such as power, may be more balanced and natural. We have observed this informally where, with head-and-shoulder shots, a projected image is presented at human scale.

Recently, we have been experimenting with projection techniques to achieve the effect of cross-table conversations. In this case, a video projection screen is placed directly against the desk, as illustrated in Fig. 5. The remote participant is then rear projected life-size. The result is powerful. The sense of presence is so strong that there is a compulsion to refer to things on the desk, despite the fact it is not really visible to the remote participant. This leads us to the topic of shared task space: what might be on the desk to discuss in the first place?

Shared Task Space

It takes very limited power of observation to note that we are sharing more than ourselves in face-to-face group interactions. I may be showing you my new sneakers, video or latest budget. Alternatively, we may both be scribbling madly on the whiteboard trying to brainstorm about the design of a new piece of software.

As there is a range of shared "accessories" and how they are used, so must there be a range of technologies in our repertoire to support similar sharing in telepresence. Like shared person space, the design space is rich and largely unexplored. The examples which follow touch the surface to give a feel for some of the issues and alternatives.

The (technically) simplest way to share some things that form part of the task space is to use the same channels as the person space. In videoconferencing, for example, we might just make sure that the subject of interest is visible to the camera. This is illustrated in the video frame shown in Fig. 6, where the participants are discussing the design of a PC board.

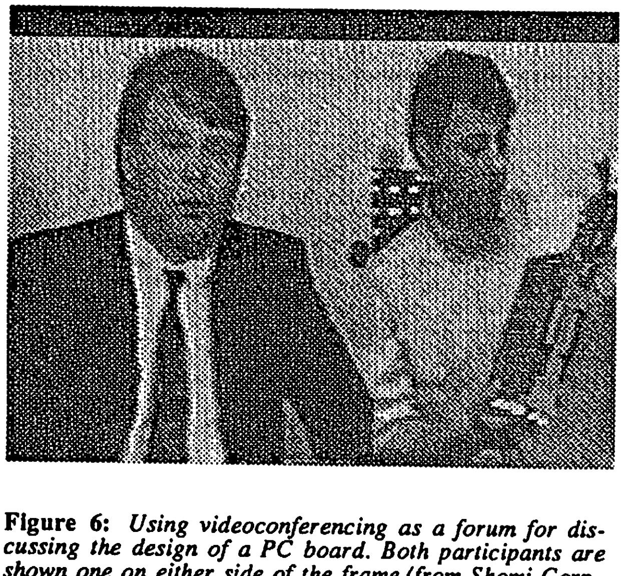


Figure 6: *Using videoconferencing as a forum for discussing the design of a PC board. Both participants are shown one on either side of the frame.(from Shomi Corp., San Diego, CA.)*

In many cases, this approach is effective and appropriate - but not always. Consider the difficulty if both participants didn't have the circuit board. Without the physical object, how would the person on the left in Fig. 6 point to problem areas, or indicate where changes should be made? While there is a *telawareness*, for the task at hand, there is clearly is not a *telepresence.*

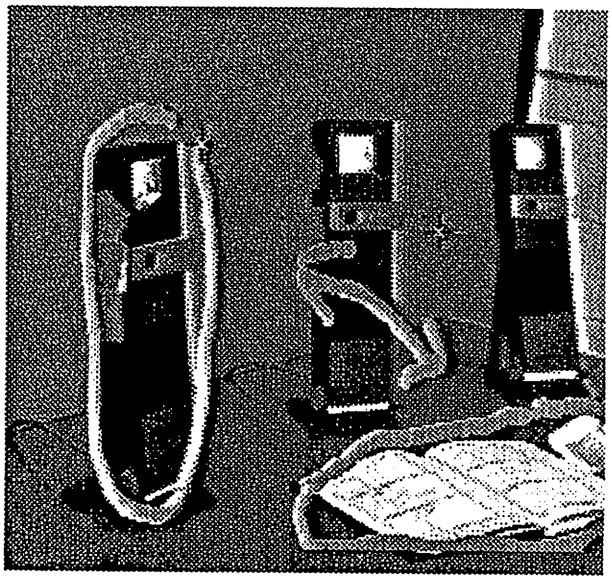


Figure 7: *Distributed shared drawing on video to enhance communications in a videoconference. Here the marking have to do with the space occupied by the Hydra units (seen in Fig. 4) and other articles on the desk.*

There are techniques that can be applied to this situation. One is a variation on a technique frequently used by television sportscasters: using a computer paint program to draw on, or *annotate* a video clip. The variation is to permit each participant in the conference to do so. This is illustrated in Fig. 7 which shows a frame from a conference where two participants are discussing the usage of the Hydra units (seen previously in Fig. 4)[4].

This technique is extended even further by Millgram and Drascic (1990). They use two video cameras mounted side-by-side (like a pair of binoculars) to capture the object under discussion. By alternating between the frames from each camera, they transmit a stereo image of the view. This is overlaid with computer-generated stereo-pair graphics (such as pointers and markers) which permits participants to work in 3D.

At a certain point, or in certain cases, however, the video channel is inappropriate for supporting shared task space. If, for example, the task was to debug some code, then it may well be more appropriate to have the software in ques-

[4] Note that the technique described differs from that found in many videoconferencing systems. In such systems, a still video image is transmitted, and one frequently cannot point at or mark-up the image. The technique described makes use of full-motion video, and may well (perhaps temporarily) use the same channel as the face-to-face communication.

tion available, rather than some video image. Here is a situation appropriate for complimenting video conferencing with shared synchronous computation.

Using dial-up telecommunications links, or computer networks, there are a number of ways that multiple users in remote sites can work together on a single computer application. A number of firms use such software, combined with teleconferencing, to provide remote product support.

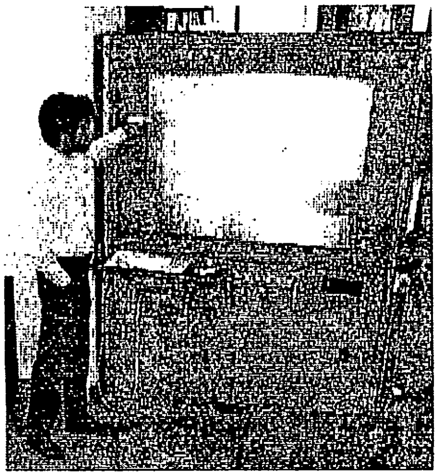


Figure 8: *Liveboard (Weiser, 1991): by combing large-screen interactive displays with advanced networks and distributed software, shared "whiteboards" can be provided to support brainstorming sessions and other collaborative work from remote sites.*

Environments such as the X window system, coupled with large interactive displays, such as Xerox PARC's *Liveboard* (Weiser, 1991) are leading towards technologies to support distributed brainstorming sessions that preserve many of the properties of same-room sessions based around a whiteboard.

What we see from the examples is that we can use a range of techniques to support both shared and person spaces, and that being able to do so is important to supporting group activity across distances. What we haven't seen - to this point - is very much on how these two types of spaces work together, or relate.

Integrating Shared Task and Person Spaces: Two Examples

Shared ARK (Smith, O'Shea, O'Malley, Scanlon & Taylor, 1990) was one of the first studies to be undertaken at Rank Xerox's Cambridge EuroPARC (Buxton & Moran, 1990). It was an investigation of joint problem solving: subjects had to determine - through the use of a computer simulation - whether one stayed dryer by running or walking in the rain. Subjects were in separate rooms. They had a high fidelity voice link and a video link, implemented using the reciprocal video tunnel shown diagrammatically in Fig. 2.

The simulation was a distributed application presented to each user on a networked workstation, and took two people to operate. Within the task space, each user was "visible" by way of an identifiable cursor in the form of a hand. The relationship of the workstation and video tunnel is shown in Fig. 9. Note that the position of the video tunnel is akin to having the remote participant sitting right beside you. Eye contact can be established by a simple turn of the head, and voice contact can be maintained throughout.

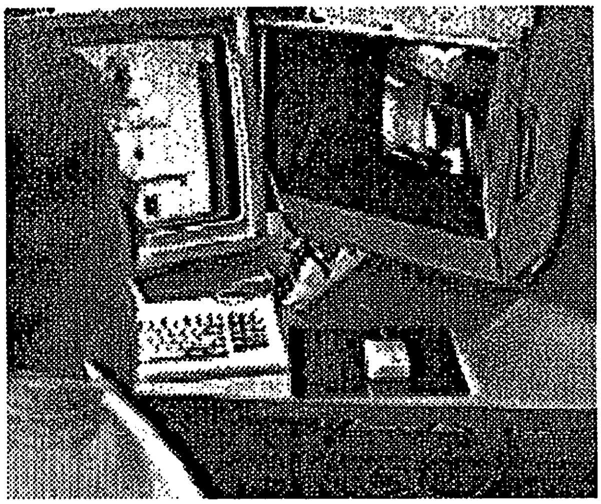


Figure 9: *Shared ARK (Smith, O'Shea, O'Malley, Scanlon & Taylor, 1990): The shared task space is on the computer display on the left. The shared person space is via the video tunnel on the left which is an implementation by William Newman of the design shown in Fig.2.*

As with working on a paper on your desk with someone by your side, you couldn't look at your collaborator's face and the computer screen at the same time. So one aspect of interest was determining when subjects visually attended to the computer display, and when they established eye contact through the video tunnel. A pattern did emerge in which eye contact was established especially when they were initially negotiating how to proceed and at the end when checking results. When actually running the simulation - which was a visually vigilant task - the video tunnel was seldom used except for short glances.

Remember, however, that the video tunnel was not the only vehicle for establishing shared person space. While attending to the computer display, each user's surrogate "hand" provided a (limited) visual personal presence through its pointing and gesturing capability. This was supplemented by the voice channel (and in a later study, Gaver, Smith & O'Shea, 1991, nonspeech audio). When visual attention was directed at the computer screen, the speech and nonspeech audio established a shared space which was more effective than the highest fidelity video display.

While what we have described is an over simplification of the experiment, it is adequate to establish that subjects moved between task and person spaces as they moved through different components of the overall task. What we take from this is the observation that some (many or most?) complex tasks require a range of channels and modalities of communication in order to be effectively supported. The reason that Shared ARK was so effective was because the methods and overhead in switching contexts (such as from computer screen to eye contact) had the same *overhead* and *action* as is used in analogous work-a-day tasks. That is, they were built on existing everyday skills that subjects already possessed, resulting in a natural behaviour. This is evident to anyone watching the experimental tapes.

Videodraw (Tang & Minneman, 1990) and its successor *Videowhiteboard* (Tang & Minneman, 1991), are excellent examples of a smooth integration of shared personal presence in a distributed task space. The systems were concerned with providing tools to support design and brainstorming activities, such as one would encounter around a drawing pad or whiteboard, respectively.

Videowhiteboard's main power came from its sensitivity to the need to support both drawing and the body language and gestures that typically accompany design and brainstorming at a whiteboard. Consequently, the system cleverly enables participants to be visible one another on the drawing surface, much like in the face-to-face situation. This is illustrated in Fig. 10, which is a frame from a video of a work session with the system.

Summary and Conclusions

Through the use of examples, we have argued that effective telepresence depends on quality sharing of both person and task space. Through this, the interaction breaks out of being like watching TV, into a direct engagement of the participants. They meet each other, not the system.

The integration of these two types of space are important. The smoothness of transitions between them is critical. Without this, the natural flow of interaction is disrupted. If the flow is to be natural, then the overhead and styles of interaction used in everyday face-to-face meetings should set the standards and design basis for telepresence technologies.

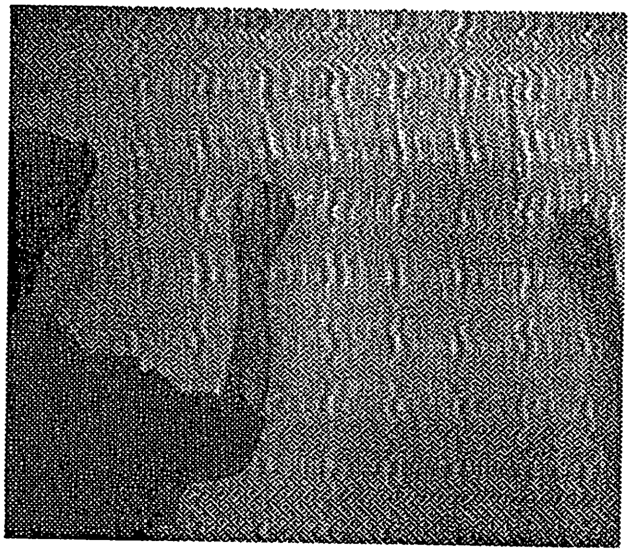


Figure 10: *Videowhiteboard (Tang & Minneman, 1991): an excellent example of effectively blending shared person and task space. The remote participant appears as a shadow on the far side of the drawing surface. The approach supports a rich vocabulary of physical gesture, including the ability to anticipate intended actions.*

What we hope the examples have illustrated is that, just as in traditional meeting spaces, one size doesn't fit all. There are a range of reasons that people meet and bonds that hold groups together. Our technologies must reflect these reasons and bonds, and their richness. Current technologies do not excel in this regard. What we hope to have shown is that this need not be so.

The design space, as afforded by available and emerging technologies, is far richer than is evident by popular practice. Hopefully the examples help show the potential and provide some keys to how it can be untapped.

Acknowledgements

This paper reflects the results of countless discussions with colleagues at Rank Xerox EuroPARC, the University of Toronto, and at Xerox PARC. This contribution is gratefully acknowledged.

Our work in this area has been supported by the Ontario Information Technology Research Centre (ITRC), the Natural Sciences and Engineering Research Council of Canada (NSERC), Xerox Palo Alto Research Center (PARC), Rank Xerox EuroPARC, Cambridge, England, The Arnott Design Group, Toronto, Apple Computer's Human-Interface Group, Object Technology International, Ottawa, Digital Equipment Corp., Maynard, MA., and IBM Canada"s Laboratory Centre for Advanced Studies, Toronto. This support is gratefully acknowledged.

References

Acker, S. & Levitt, S. (1987). Designing videoconference facilities for improved eye contact. *Journal of Broadcasting & Electronic Media*, 31(2), 181-191.

Buxton, W. & Moran, T. (1990). EuroPARC's Integrated Interactive Intermedia Facility (iiif): early experience, In S. Gibbs & A.A. Verrijn-Stuart (Eds.). *Multi-user interfaces and applications,* Proceedings of the IFIP WG 8.4 Conference on Multi-user Interfaces and Applications, Heraklion, Crete. Amsterdam: Elsevier Science Publishers B.V. (North-Holland), 11-34.

Buxton, W. & Sellen, A. (1991). Interfaces for multiparty video conferences. University of Toronto. Submitted for publication.

Fields, C.I. (1983). Virtual space teleconference system. *United States Patent 4,400,724,* August 23, 1983.

Gaver, W., Smith, R. & O'Shea, T. (1991). Effective sounds in complex systems: the ARKola simulation. *Proceedings of the 1991 Conference on Human Factors in Computer Systems, CHI '91,* 85-90.

Ishii, H. & Miyake, N. (1991). Toward an open shared workspace: computer and video fusion approach of TeamWorkStation. *Communications of the ACM,* 34(12), 37-50.

Ishii, H. & Kobayashi,M. (1992). Clearboard: a seamless medium for shared drawing and conversation with eye contact. To appear in the Proceedings *of CHI '92,* May 1992.

Kraut, R. & Egido, C. (1988). Patterns in contact and communication in scientific collaboration. *Proceedings of CSCW '88,* 1-12.

Millgram, P. & Drascic, D. (1990). A virtual stereographic pointer for a real three dimensional video world. In D. Diaper et al. (Eds), *Human-Computer Interaction - INTERACT '90.* Amsterdam: Elsevier Science Publishers B.V. (North-Holland), 695-700.

Rosenthal, A.H. (1947). Two-way television communication unit. *United States Patent 2,420,198,* May 6, 1947.

Sellen, A. (1992). Speech patterns in video-mediated conversations. To appear in *The Proceedings of CHI '92*, May 1992.

Sellen, A., Buxton, W. & Arnott, J. (1992). *Using spatial cues to improve desktop video conferencing*. 8 minute videotape. To appear in the *CHI '92 Video Proceedings*.

Smith, R., O'Shea, T., O"Malley, C., Scanlon, E. & Taylor, J. (1990). Preliminary experiments with a distributed multi-media, problem solving environment. Unpublished manuscript. Cambridge: Rank Xerox EuroPARC.

Tang, J. & Minneman, S. (1990). Videodraw: a video interface for collaborative drawing. *Proceedings of the 1990 Conference on Human Factors in Computer Systems, CHI '90*, 313-320.

Tang, J. & Minneman, S. (1991). Videowhiteboard: video shadows to support remote collaboration. *Proceedings of the 1991 Conference on Human Factors in Computer Systems, CHI '91*, 315-322.

Weiser, M. (1991). The computer for the 21st century. *Scientific American*, 265(3), 94-104.

CSCW - WCSC: Computer-Supported Cooperative Work - What Changes for the Science of Computing

Marilyn M. Mantei

Dynamic Graphics Project
Computer Systems Research Institute and
Department of Computer Science
University of Toronto
6 Kings College Road
Toronto, Ontario, Canada M5S 1A4
416-978-5512 FAX: 416-978-4765 E-mail: mantei@dgp.utoronto.ca

Abstract

Computer-supported cooperative work environments change some of the underlying beliefs about solutions that have been built for distributed computing. Electronic mail and file transfers have worked efficiently and effectively by breaking the information to be transferred into packets and reassembling the packets at the destination. This is not a viable solution for handling shared real time drawing or writing. Integrity has been maintained by locking out portions of a database until an update is completed. Such lockout is not always suitable in a groupware interface. Other solutions are necessary for resolving conflicts. Client-server architectures have worked well for managing distributed processing but replicated architectures, coupled with their fragility and synchronization problems are needed for groupware products in order to preserve acceptable local response time. The addition of multi-media to the environment complicates the problem more. Control of analog video requires a client-server environment which can build in intolerable delays as distances between communicating parties increase. This paper approaches design criteria for shared software from the human side and points out profound architectural problems that need to be solved if multi-media cooperative work environments are to function effectively.

Resumé

Les environments de tâches coopératives médiatisés par ordinateur changent certaines des croyances sur les solutions qui ont été mise en oeuvre pour l'informatique distribuée. Le courrier électronique et le transfert de fichiers ont fonctionnés avec succès et efficacité en séparant l'information en paquets qui sont réassemblés a` l'arrivée. Cette solution n'est pas viable pour traiter la composition coopérative de textes ou de dessins en temps réel. L'intégrité a été maintenue en bloquant l'accès de portions de base de données jusqu'à la fin de leurs mise à jour. De tel blocages ne toujours conviennent pas pour une interface de informatique de groupe. Des solutions nouvelles sont nécessaires pour résoudre ces conflits. Les architectures client-serveur fonctionnent bien pour gérer des processus distribués. Mais des architectures répliques, avec leurs fragilité et leurs problèmes de synchronisation, sont nécessaires aux produits groupware afin de préserver les temps de réponse locaux acceptables. L'addition de multi-média complique encore le problème. Le contrôle de la vidéo analogique demande un environement client-serveur qui crée des délais intolérables lorsque les distances entre interlocuteurs augmentent. Cet article présente des critères de conception de informatique de groupe d'un point de vue utilisateur et fait apparaître des problèmes architecturaux profonds qui doivent être résolus si l'on veut permettre aux environements coopératifs multi-média de fonctionner de manière efficace.

Introduction

The 1980's were the decade of the personal workstation. The 2000's will be the decade of computer-supported cooperative work (CSCW), in which shared windows will open on multiple workstations across hallways and even across continents (Baecker, 1991). Video images of co-workers in both real and delayed time will complement the shared windows. Shared work environments and multi-gigabyte fiber transmissions will be commonplace. Audio connections are moving to the airwaves (cellular telephones) while more intensive data transfer (video conferencing and shared work) is taking over the high bandwidth networks (Dertouzos, 1991). As exciting and imminent as these changes are, implementing CSCW environments requires an essential rethinking of workstation hardware, distributed systems, networks and data transfer to support the new environments.

People are real time systems. Much of today's distributed processing solutions have been built on the assumptions that delays do not matter and that packet switching and asynchronous tranfer of information is acceptable (Tanenbaum, 1992). When the distributed human processor is added to the system, these assumptions are no longer valid. With individual interfaces, certain tolerances of delays and non-synchronicity of events were accepted because the work being accomplished was private. In cooperative environments, work is performed jointly and interaction requires each individual to exhibit a range of subtle behaviours for negotiation, teambuilding, expressing doubt, etc. What worked before won't work now: the social cost is far too great.

The above paragraph characterizes the nature of the constraints facing the builders of cooperative work environments. The constraints are what I call "deep" system problems, i.e., their solutions are based on the very architecture and hardware on which the system is built. The constraints are also human interface problems, i.e., they are brought on by the needs of the human user.

A common misconception in computer science is that the user interface design can be separated from the working part of a system. This misconception has led to the development of user interface toolkits and user interface management systems for supporting the design process. Although these tools have been effective in making some aspects of designing interfaces easier, they can blind the designer from perceiving more functionally based difficulties with the interface design. In CSCW, it is clear that these tools are not enough and that more fundamental changes to the underlying system are necessary if the interface is to work. Thus, the basic premise of this paper is that CSCW is one human interface area that needs to reach deep into the computer science field for support. Tools for manipulating screen objects and handling input devices no longer suffice.

Why is the design of cooperative work interfaces so challenging? Humans are social animals and have very finely tuned skills for handling social contact. If the tool they use affects their ability to project themselves or to maneuver comfortably in the environment created by the tools, the entire tool will be rejected. In contrast to individual interfaces, where the very privacy of working at the interface protects the person learning the interface from ridicule, the public nature of shared interfaces makes users extraordinarily sensitive to small problems.

The sections which follow in this paper take a human-centred point of view in examining the design issues in building CSCW systems. Human needs are stated and explained. What these needs translate to in terms of underlying system design is then discussed along with a presentation of various solutions that have been tried and the problems that have been encountered. A major difficulty with constructing an architecture that fulfills one human need of a CSCW system is that the architecture then violates another need. Often all current design solutions are found inadequate and in need of new research to make the cooperative system work appropriately.

A major focus throughout the discussion is the assumption that thousands of users will someday be working with the system so that scalability, flexibility, robustness, and support for workstation heterogeneity are important issues (Arango, et al., 1992). Many solutions for today's prototype systems, which manage 15 to 100 connections, are not viable for interoffice connections of most major corporations or government organizations which may involve thousands of simultaneous connections.

For my discussion I select two representative CSCW technologies. I begin with desktop video conferencing systems (also called media spaces), which represent those technologies designed to allow people to meet visually while still remaining a long distance from each other. This technology is invariably connected to computing technology which supports the video and audio connections but also provides some form of shared software. The interface needs for supporting shared software are treated in the next section.

There are many aspects of CSCW that I do not discuss. They include such items as shared calendars, hypertext systems, office coordination systems, video mail, group decision support systems, etc. Each of these applications is either asynchronous - and therefore does not make the demands on the distributed processing that synchronous linkage entails - or poses problems that are the same as those arising from the desktop video conferencing and shared software.

This paper is a discussion of problems not solutions, but it is one with a hopeful note. It provides a new set of constraints that help to focus the design process and suggests rethinking the mechanisms that are used to handle distributed processing and video network traffic. It suggests that solutions can be found and often points to potential paths to take especially in the real time systems area.

Desktop Video Conferencing

Desktop video conferencing involves the interconnection of offices which can be relatively far apart (e.g., in different cities) by a complete video and audio hookup (Watabe, et al., 1990; Crowley, et.al., 1990). It has some similarities to a videophone, in which the two parties in a long-distance conversation have a visual picture of each other in addition to audio. It is also more than a videophone because it supports more services, e.g., meetings, browsing, shared common areas, etc. Desktop video conferencing systems are designed to support the ubiquitous visual contact that facilitates collaboration in day-to-day communication (Kraut, Egido & Galegher, 1988). History has demonstrated a very poor market for videophone calls and video conferencing (Egido, 1988), but current research with media spaces (Goodman & Abel, 1986; Buxton & Moran, 1990) has indicated effective casual communication uses that the visual channel supports. Figure 1 illustrates a typical configuration of the type of network infrastructure that supports desktop video conferencing.

Today's desktop video conferencing architectures assume different channels of communication for the different media. The video is shipped via coaxial cable or fiber and the audio is sent either by normal telephone connections or by separately mixed audio. The computer controls switch boxes which make the connections. Tomorrow's A/V communication will be digital but still shipped by physical connections, e.g., dark fiber. This makes the soft switching of the connections an easier problem, but each type of communication must still be handled uniquely (Vin et al., 1991). This is inherent in the nature of the information being shipped. Audio for multi-person contacts needs to be mixed separately to filter out noise from each site. Video requires compression to ship. Effective compression strategies are hybrids of techniques

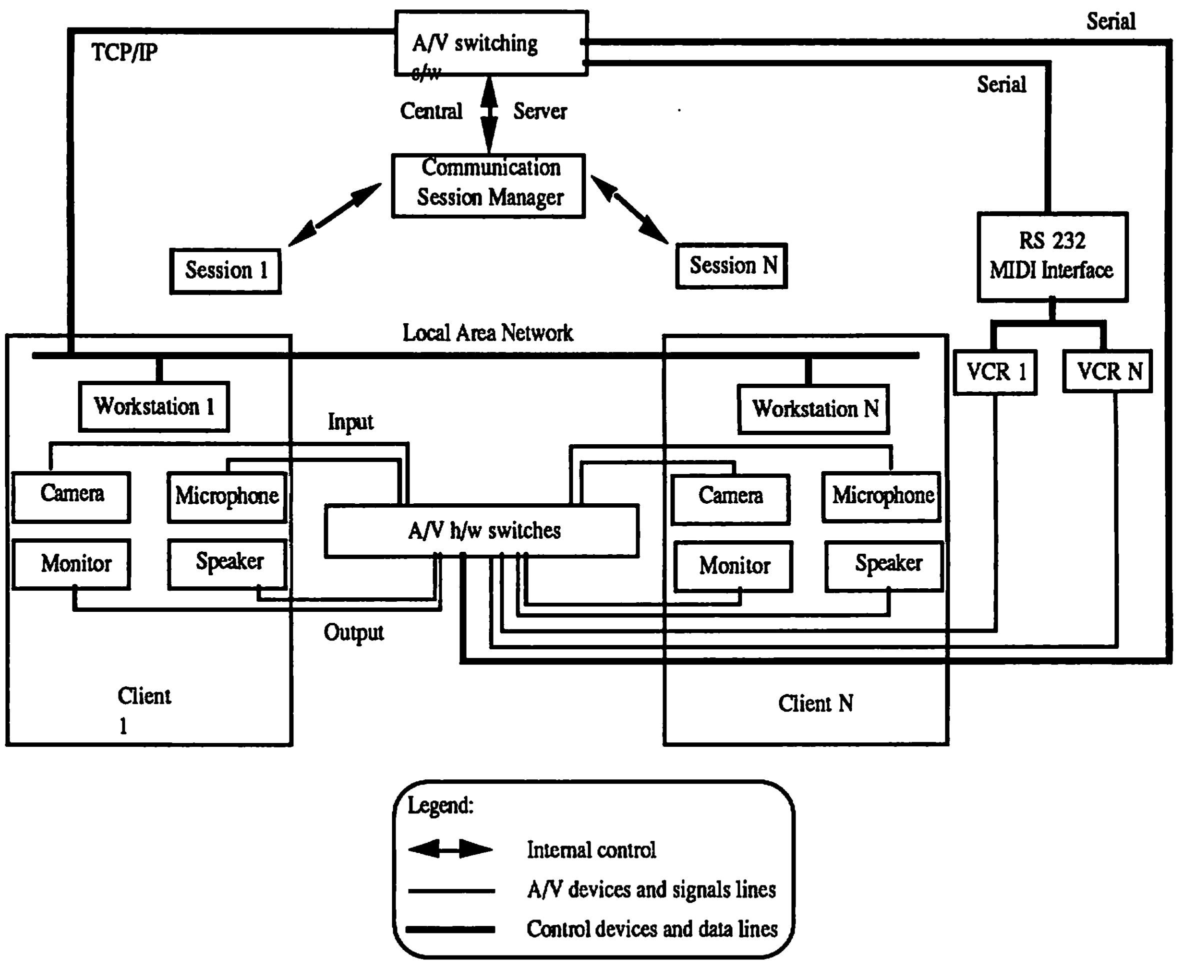


Figure 1 Typical configuration of current desktop video conferencing configurations. The video and audio are shipped by analog and a local area network passes the control informaton to a central server which is manageing the switching..

based on changes in the visual scene over time and information present in the visual scene (Fox, 1991). Thus,scenes from the different sites in the multi-person conversation will be compressed and managed differently. Exchanges of computer messages handling the shared workspaces will form the third channel. In addition to managing shared computer workspaces, the computer signals on this channel will also be used to manage the transmission of the audio and video channels, e.g, authorizing an additional person to enter into a video meeting.

Current desktop video conferencing installations ship the audio and video over analog channels and use a client-server architecture to manage the switching (Buxton & Moran, 1990; Arango et al, 1992). The client computer is located close to the switching centre and maintains a database of connection states to manage the A/V switching. Digitized video and audio transmission need not be controlled by a centralized switching arrangement opening up other possible control architectures.

In the discussions of the user interface to desktop video conferencing which follow, digitization of all signals is assumed. This opens the possibility of comparing client-server versus distributed A/V controls from the standpoint of user needs for this environment. This also opens up the issue of packaging the different varieties of digitized information that are to shipped plus a much larger issue, that of shipping the very large amounts of data arising from digital video. Current network solutions have not been designed for this type of traffic. The list of user issues associated with video conferencing is endless. Below, I discuss six important ones to illustrate the major impact of network architecture on the desktop video conferencing interface.

Modality Synchronicity

Humans communicate by many modes. They talk, they gesture, they point, they look at things with their eyes. What they are relatively unaware of is how synchronized these activities are. Gestures end at the end of a clause, eyes change direction at sentence end. Not having this synchronicity is symptomatic of underlying brain disorders. The problem is a variation of motor aphasia, and it is very disturbing for others to engage in communication with a motor aphasic person. Despite a feeling of deep uneasiness, the recipient of the asynchronous communication cannot describe what is wrong with the conversation. An example that most of us are familiar with is the TV character, Max Headroom.

When voice and video signals are captured and shipped separately, it is likely that they will arrive slightly out of synch. Such loss of synchronization is worse than a badly dubbed movie because it destroys all the redundant cues the listener relies on for understanding the conversation. If, in addition, the individual is heard to be typing or seen to be drawing in the video picture, and the screen update information arrives late or early, the flow of discussion proceeds haltingly as each person waits for corroboration that all information has arrived. Early video conferencing systems often used a speaker phone to send the audio portion of the signal, but quickly found this setup to be inadequate. Human speech is highly overlapping (Buxton & Sellen, 1992), but speaker phone technology does not permit this overlapping. Without the visual channel, attempts to speak are not apparent. With the visual channel, the limitations of the speaker phone are all too visible.

Desktop video conferencing systems can be designed to pack and ship all signals as one packet, but this leads to other problems. Audio is very different from video and demands different handling characteristics. Although both are continuous signals, the sampling rates can be very different. Screen events will also have different closure times than either video or audio signals. A series of mouse selections taking 500 milliseconds or more may occur before it is possible to send a valid user event. Whether the audio, video and user event information is sent digitally down a single channel to be unpacked by the receiving workstation or down separate channels, synchronous display of all channels of information is essential for the success of the system.

Entering and Leaving Sessions

Unlike telephone contact, visual contact is rarely restricted to two people in an active workplace. Sighting other individuals working together in an office is often an invitation for a third and possibly fourth person to join the conversation (Root, 1988). Video sessions need to support similar behaviour. This implies a video conferencing capability in which participants can scan all video conversations and ask to join those which interest them. It also implies an environment in which people can leave a video conversation and new people can join maintaining a continuous thread of connected people even after both originators of the conversation have left.

A variety of mechanisms exist for handling multiple individuals in a video meeting. Most arrangements mix the video signal at a centralized source and ship the merged version down a single video channel to each recipient. Compressing the video signal makes mixing more difficult. Nevertheless, this can be done at a communications bridge which unpacks each signal, mixes it and then compresses it again for shipping. An inherent disadvantage is incurred in having to unpack the video signal twice. Mixing the video at a centralized site makes it easier to build interfaces that allow others to peer into meetings that are going on but centralized mixing is an architecture that does not scale up very well. Centralized video mixing also means that a software bridge is needed for each meeting that is taking place, even for two person meetings because of the potential for adding more participants to these meetings.

Video signals can alternatively be sent directly to each workstation and mixed there, but this implies that each workstation have the ability to handle that level of input traffic. Since digitized video signals already require high bandwidth to ship, this is currently not a feasible arrangement.

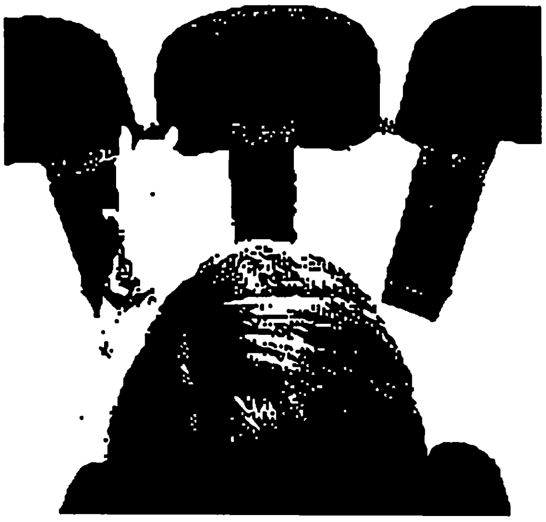


Figure 2 A user is seated in front of three Hydra units. In the photo, the Hydra units sit on the table in front of the chairs that would otherwise be occupied by three remote participants. Each Hydra unit contains a video monitor, camera, and loudspeaker. A single microphone conveys audio to the remote participants. (From Buxton & Sellen, 1992)

Hydra is an example of video meetings that are not mixed at a central site (Buxton, 1991). It is one of the more innovative interfaces that solves the problem of audio and positional location in video conferences. Hydra provides a video image, camera, speaker and microphone in a small desktop unit for each person engaged in the meeting. If four people are meeting, a Hydra user would have three units on the desktop, each one representing a virtual person in the meeting. Figure 2 contains a diagram of one of the Hydra units. Hydra currently works by shipping video signals down separate channels. This type of transmission uses an enormous bandwidth. Hydra also has no mechanism for displaying a visual picture of the meeting taking place to others who might want to join the meeting. Although Hydra solves important social considerations for meetings, its implementation in a large scale communication system would be very difficult

Maintaining Continuous Sessions

Our research at the University of Toronto has shown that one of the very viable modes of a desktop video conferencing environment is an open channel (Li & Mantei, 1992). A visual channel is maintained continuously between one or more offices to support ad hoc information exchange by people working closely together. These connections are maintainable with the architecture shown in Figure 1 but do not handle a private video call. A private call is different from a meeting. The caller wants to talk to one person only. Such a setup is equivalent to having a two-party line, but the line needs to support the visual contact for both parties yet not connect all other parties into a video conference. The setup also needs the capability of connecting the parties into a video conference, if at some point, the caller wants to join the larger group. Thus, the underlying connection architecture needs to be able to switch the caller to the software bridge which is maintaining the other connections

Privacy Considerations

Up until now, we have only discussed the problems of making different multiple connections and have not addressed the issue of preventing users of the system from making visual contact. It is believed that being able to

view co-workers at work provides useful opportunities for communication which co-workers take advantage of (Kraut et al., 1988). Limiting the video channel to a meeting or a video call prevents these visual opportunities yet keeping an open visual channel infringes on individual privacy. What is needed is a system that is both designed to permit large amounts of visual browsing but that also allows participants to limit browsing intrusions by giving browsing rights to a subset of people. The underlying design issue is where to keep the privacy informaiton, locally or in a centralized database.

Privacy settings can be kept at each desktop node, but if people are browsing meetings, then the privacy considerations of each person in the meeting need to be handled. Such privacy screening can cause long delays for larger meetings where nodes in many different cities require checking. A faster way is to manage meetings at a centralized location with privacy being the lowest common denominator of the assembled group.

Soft Communication Failures

A common problem with long distance communication technology is one of providing alternatives to communication failures. The basic failure is one of not reaching the individual called. Voice mail and the telephone answering machine are solutions to these problems. When direct contact fails, there is a backup storage device for leaving an audio message. This works well for voice communications, but video contact because of its much larger bandwidth and cost of storage may not be amenable to the same solutions. Currently, voice mail storage is centralized. Callers record their message and have options of replaying it and re-recording it if it sounds unsatisfactory. Users can then call the central storage and access their calls. This setup works reasonably well for voice messaging. How would it work for video messaging?

A caller would send a video message to a centralized site. This would be digital video involving a very large amount of storage. If the caller wanted to play back the message, the digital video would be sent back to the caller. This is a large volume of network traffic which can be avoided by local playback and shipping when the message is completed. However, local playback requires local storage which is currently expensive for digital video. This problem can be cheaply bypassed by storing analog video which can be digitized when the message is finally ready to transmit. Local video storage is also prone to failure which implies additional backup strategies that use (1) other local storage devices or (2) a centralized device when all else fails. The underlying argument is that mechanisms for supporting transitions to video mail cannot be handled by using the current storage solutions for telephony.

Support for Multiple Contacts

Video wears two hats in the office. It is a broadcast medium and a meeting tool. This implies support for both functions in the desktop video conferencing environment. It also implies the capability to move between either function with little difficulty. Broadcast video involves a multipoint setup that distributes a single video image to many sites but does not necessarily receive video images from these sites. Broadcast connections are often sent to interrupt other connections such as the audio landing notifications on airplanes which interrupt music channels, but broadcasts can often be secondary information that users would like to see interruptible by video phone calls. A network which manages broadcasts is configured differently than a network which handles point to point contacts. In a desktop video conferencing environment, both activities need support.

Activity Sharing Software

Activity sharing applications are programs in which multiple individuals working on separate workstations can simultaneously be viewing and updating the same work. These workstations can be a few metres or hundreds of kilometres apart. Messages sent along networks joining the workstations update the screens with each users input. A variety of such programs exist for different workstation platforms and some are commercially available such as Aspects^{tm} (Group Technologies, 1991). Shared activity applications primarily support shared writing and shared drawing tasks (Fish, et al., 1988; Tang & Minneman, 1990, Greenberg & Bohnet, 1990), but prototypes also exist for supporting programming (Brothers, Sembugamoorthy & Muller, 1990) and database design work (Hayne & Ram, 1990). Most software has been written as application specific, e.g., for wordprocessing only, but some work has been done to build an underlying operating system or window system that will support a wide variety of single user applications (Lauwers & Lantz, 1990; Lauwers et al., 1990).

Two main types of architecture support the shared activity *sessions.* The first is a client-*server* architecture in which the application resides on a designated workstation and all other workstations send messages to update the data structure *residing* in the application (University of Michigan, 1990). The application, in turn, sends out messages to update the screens on each workstation. The second is a replicated architecture. (Replicated architectures are also called serverless or peer-to-peer architectures.) In the replicated architecture, an application resides on each workstation and exchanges messages with all other workstations, accepting input generated by other users and sending input to update the screens of the other users (Crowley et al., 1990). Replicated architectures have serious fragility problems (Ahuja, Ensor & Lucco, 1990) because of difficulties with synchronization and heterogeneity in workstations whereas client-server architecture generate a large amount of message traffic and have to maintain states of all workstations. Hybrids of partially replicated and part client-server architectures also exist (Mawby, 1991; Lu, 1992) which bring the advantages and disadvantages of both solutions.

From the human perspective, either solution has problems in supporting collaboration. Instead of either extreme, the hybrid between the client-server and replicated architecture needs to be tuned very carefully to meet the user constraints listed in the sections that follow. The shared architectures need to be concerned with maintaining adequate system response time, adding latecomers to a work session, not disrupting the flow of work with lockout procedures, synchronizing the work between collaborators, etc.

Response Time

A user who is developing drawings using a mouse or a stylus interface needs to have the immediate feedback on the results of their motor behaviour displayed on the screen if they are to use their finely tuned hand-eye coordination capabilities. Lags in displaying figures on the screen are unacceptable and make the drawing task extraordinarily difficult. Current client-server arrangements which send the user input to the server and then back to the client are far too slow for this microsecond coordination.

Users also need to see the drawing behaviour of their collaborator in real time. If their collaborator is drawing a box or circle or selecting text to delete, the size and placement of of the figure or the span of text selected show up on the screen as feedback to the user before a key is released or other action taken to indicate acceptance of the sizing, placement or text selection outlined on the screen. If these events are not shown in real time to the collaborator it is difficult for the collaborator to comment on the work being done (Lu & Mantei, 1992). A client-server architecture could synchronize these times better than replicated architectures especially if distances between sites were large, but both architectures will have problems with synchronizing this continuous event traffic. This is a real time concern.

Hand-eye coordination is important to support in shared wordprocessors as well as shared drawing tools. Text that is delineated by a mouse controlled cursor is a variation of drawing behaviour. Gesturing and pointing on text require similar hand-eye coordination and thus, real time support.

Adding Latecomers

Meetings rarely begin or end with all people present. Attendees join sessions late and others leave early. This requires updating the newcomer to what has transpired in the meeting and making sure that the early leaver has a final document of all the decisions made at a meeting.

In a shared activity session, the problem of updating the workstation of a latecomer is a different task depending on whether a client-server or a replicated architecture is used for managing the shared session (Chung, 1991). With a client-server architecture, the problem is one of setting the window parameters of the newcomer and updating the shared session information of the other participants. With a replicated architecture, the problem is one of transferring a copy of the work to the newcomer's workstation. On the social behaviour side of the issue, the problem is one of integrating the newcomer into the meeting with little disruption. The newcomer needs to obtain a copy of the current state without interrupting the flow of changes and additions that are being made to the work product. Currently, most replicated architectures lock out all work until the newcomer's workstation is updated. Others simply do not permit latecomers. Client-server architectures such as Rendezvous (Patterson et al., 1990) can support a more graceful entry but run into problems in providing an accurate replication of other user's workstation environments. This problem becomes particularly acute in heterogenous workstation environments. Although the client-server architecture is much better at handling the latecomer problem, the effects of a client-server architecture on response time latency strongly support a replicated solution.

Lockout and Concurrency Issues

Up until recently, users have not had the capability to work in the same space at the same time. Shared writing, drawing and spreadsheet software give us this functionality but not without integrity concerns. If one user deletes a sentence at the same time another user is modifying it, what is the end result? One solution is to prevent modification access to any object that is already in use by another user (Ellis & Gibbs, 1988), but this creates new problems. To be locked out, text or figures need to inform all workstations of their modification status. Both the informing process and the interrogation process take time. Users neither like waiting for access to a screen object (e.g., a word) before they can use it or being denied access seconds after they have attempted to modify the screen object. The denial of access is a particularly acute problem because users have already begun their cognitive work plan when they attempt to select the object (Card, Moran & Newell, 1983). Users want to know in advance when text or figures are locked out to avoid unproductive mental effort.

In most writing tasks, concurrency is not an issue. Lockout is usually at the character level as is seen in such shared editor systems as Aspects (Group Technologies, 1991), ShrEdit (University of Michigan, 1990), and

SASE.(Mawby, 1991). With such a fine resolution for lockout, the probability of concurrent access is low. Writing is such a complex cognitive process, that people composing text prefer to do it on their own (Posner & Baecker, 1991). However, one of the major times when users are likely to collide occurs frequently. When one participant in a shared session is typing an idea and others are observing its generation, typing or spelling errors occur which other users jump in to fix. They cannot perform these correction because this text is locked during creation.

Ellis, Gibbs and Rein (1988) give a good presentation of alternate solutions to lockout which support a more flexible participant environment. One of their suggestions is that of allowing individual work to diverge and putting the onus of the repair on the participants. Another possibility, especially for drawing tasks, is to ignore concurrency and to throw away messages that request changes to objects that no longer exist. This is done in CaveDraw (Lu, 1992) where one user can delete the very layer another user is drawing on. Collision events are assumed to be relatively infrequent and negotiable by the other multi-media support available. For other shared software tasks, similar optional concurrency control measures can be in place. This implies that the software which controls concurrency issues will need to be smarter and know when and where concurrency will be problematic.

View Synchronicity

Users of collaborative tools often do not use them in a collaborative fashion. There are collaborative tasks that people perform that are so complex, e.g., writing or programming, that interacting with others inhibits the performance of the task (Neuwirth, 1990). Often collaborative work sessions are a combination of people working together, parceling out the work to be done and then working individually on the work. The advantage of a collaborative work environment is that it allows co-workers to smoothly move into and out of collaborative mode. This type of observed movement between collaborative and non-collaborative states is common in computer supported meeting rooms (Mantei, 1988).

The underlying application support for allowing both types of work patterns in a collaborating group will permit participants to have different views of the work product. It also will have a functionality for synchronizing participant's views. Synchronized views work in tandem. When one participant scrolls to a new place in the text, the other participants see their windows scroll as well. To synchronize views, the software needs to maintain information on the screen states of each of the participating workstations. Since individual users may have differing window sizes, use different fonts and even have different objects occluded, synchronization requires a large amount of mapping operations between each workstation.

Since a sequence of user events at one workstation is often necessary to determine the synchronization at the other workstation, arbitrarily breaking up the events into message packets can destroy this context. For example, a user can scroll to a position and then select the visible text. The workstation that is synchronized with the first workstation can scroll to an approximate location, but if text is selected in a part of the window on the first workstation that is occluded on the second workstation, the second workstation will have to execute a second scroll and then show the text selection. This type of synchronization looks jumpy and awkward to the participants, not to mention the concern that both participants are *still not* seeing the same text.

Occlusion in many window systems may be difficult to detect because one of the fundamental premises of window systems is that applications should be able to write into windows without concern for what portions of the window are occluded (Scheifler, Gettys & Newman, 1988). Without control of occlusion, view synchronization becomes a difficult problem.

Synchronization is also one case where *concurrency* control is necessary. Once screens are synchronized, scroll bar usage can lead to "scroll wars" as can other global events which update the screen (Stefik et al., 1987). So, although synchronization is very much at the interface level, mechanisms that pass the streams of user behaviour between the synchronized workstations also have to examine these event streams and either require homogeneity in the resources used by the participants or interpret the streams to best represent a synchronized environment.

Gesture Support

When people work together, they use their hands to point, circle,or motion in a wide variety of ways about the information that is being created (Tang, 1989). Support for gesturing in shared activity software comes in the form of a telepointer, that is, a cursor which is seen on everyone's screen that is controlled by the owner of the telepointer. Each participant in a shared work session has a personal telepointer, usually uniquely identified by shape or colour. Each participant also has their own cursor for moving around their workscreen.

Workstations therefore need to support multiple cursors, i.e., the cursor of the person at the workstation and the cursors of any of the participants that are in telepointer mode. Since cursor functioning is done at low levels in systems software and since most workstations are rarely equipped to handle more than one mouse input, this support suggests fundamental changes in workstation and/or operating system design. Workstations will need to support multiple cursors, and if necessary, to change the shape and form of the cursors as they move over the windows (Greenberg, 1991).

Version Control

In a replicated architecture, which workstation has the most recent version? Is it the one that left the session last, the one with the latest time stamp, or the one that has been designated to hold the latest version? If an individual works alone on the work product after the session, is the update automatically transferred to each of the other workstations? What happens if two people work on the file separately but at the same time? Storage mechanisms need to be put in place that allow multiple users to maintain version control of the work product. This is not a hard technological problem. What is necessary is that the solution be incorporated in the basic architecture of the system rather than at the application level. Otherwise the latest version might reside on a workstation that is not in operation or connected to the next joint work session. Humans are notoriously bad at version control. It is best to leave this task to the system.

Conclusion

In this paper I have discussed a set of requirements that need to be met if the social nuances of collaboration are to be supported by cooperative work tools. What is important about these requirements is that the ability to meet them lies in solutions at the very heart of the underlying system architectures and communication structures that support CSCW. Small adjustments at the interface level will not fix the problems. In some cases, only redesign of the computer workstation and/or its operating system will make the problem solutions doable Some of the most viable fixes for one requirement contradict the most viable solution for another. Some of the solutions violate current ways of thinking about the world, i.e., we don't need to guard against concurrency. I haven't presented solutions to these problems. I am not the systems guru. However, demonstrating a problem's existence is always the first step in its solution.

Acknowledgements

For research support the author is indebted to the Natural Sciences and Engineering Council of Canada (NSERC), the Information Technology Research Centre of Ontario (ITRC), Apple Computer, Inc., Rank Xerox EuroPARC, Digital Equipment Corporation, IBM Canada's Laboratory Centre for Advanced Studies and Alias Research. I also wish to thank Catherine Plaisant for translating the abstract into French, Kevin Schlueter for his many insightful comments on the manuscript and Jin Li and Gifford Louie for their artwork.

References

Ahuja, S. R., Ensor, J. R., and Lucco, S. E. (1990) A Comparison of Application Sharing Mechanisms in Real-Time Desktop Conferencing Systems. *Proceedings of COIS'90 Conference on Office Information Systems*, Cambridge, MA, April 25-27, 1990, pp. 238 - 248. New York: ACM Press.

Arango, M., Bates, P., Fish, R., Gopal, G., Griffeth, N., Herman, G., Hickey, T., Leland, W., Lowery, C., Mak, V., Patterson, J., Ruston, L., Segal, M., Vecchi, M., Weinrib, A. and Wuu, S. (1992) Touring Machine: A Software Platform for Distributed Multimedia Applications, *Proceedings of 1992 IFIP International Conference on Upper Layer, Protocols, Architectures and Applications*, Vancouver, BC, Canada, May 1992.

Baecker, R. (1991) New Paradigms for Computing in the Nineties. *Proceedings of Graphics Interface'91*. Calgary, AB, Canada, June 1991, pp. 224 - 229.

Brothers, L., Sembugamoorthy, V. and Muller, M. (1990) ICICLE: Groupware for Code Inspection. *Proceedings of CSCW'90 Conference on Computer Supported Cooperative Work* , Los Angeles, CA, October 7 - 10, 1990, pp. 169 - 181, New York: ACM Press.

Buxton, W. A. (1991). Telepresence: Integrating Shared Task and Personal Spaces. *Proceedings of Groupware'91*, Amsterdam, Holland, October 1991.

Buxton, W. A. and Moran, T. P. (1990) EuroPARC's Integrated Interactive Intermedia Facility (iiif): Early Experience. *Proceedings of the IFIP WG 8.4 Conference on Multi-user Interfaces and Applications*, Heraklion, Crete, pp. 11 - 34. In S.Gibbs and A. A. Verrijin,-Stuart (Eds.) *Multi-User Interfaces and Applications*. Amsterdam: North-Holland.

Buxton, W. A. and Sellen, A. J. (1992). Interfaces for Multiparty Video Conferences. *Proceedings of CHI'92 Conference on Human Factors in Computing Systems*, Monterey, CA, May 3-5, 1992. New York: ACM Press.

Card, S. K., Moran, T. P., and Newell, A. (1983). *The Psychology of Human-Computer Interaction*. Hillsdale, NJ: Lawrence Erlbaum.

Chung, G. (1991). *Accomodating Latecomers in a System for Synchronous Collaboration*. Unpublished masters dissertation, Department of Computer Science, University of North Carolina, Chapel Hill, North Carolina, August 1991. Available as report no. TR91-038.

Crowley, T., Milazzo, P., Baker, E., Forsdick, H. and Tomlinson, R. (1990). MMConf: An Infrastructure for Building Shared Multimedia Applications. *Proceedings of CSCW'90 Conference on Computer Supported Cooperative Work* , Los Angeles, CA, October 7 - 10, 1990, pp. 329 - 342, New York: ACM Press.

Dertouzos, M. L. (1991). Communications, Computers and Networks. *Scientific American* 265(3), September 1991, pp. 62 - 69.

Egido, C. (1990). Video Conferencing as a Technology to Support Group Work: A Review of its Failures. *Proceedings of CSCW'90 Conference on Computer Supported Cooperative Work* , Los Angeles, CA, October 7 - 10, 1990, pp. 13 - 25, New York: ACM Press.

Ellis, C. A., Gibbs, S. J. and Rein, G. L. (1988). *Groupware: The Research and Development Issues*. MCC Technical Report Number STP-414-88, Austin, TX.

Ellis, C. A. and Gibbs, S. J. (1988). *Concurrency Control in Groupware Systems*. MCC Technical Report Number STP-417-88, Austin, TX.

Fish, R., Kraut, R., Leland, M. and Cohen, M. (1988). Quilt: A Collaborative Tool for Cooperative Writing. *Proceedings of COIS'89 Conference on Office Information Systems*, Palo Alto, CA, March 23-25, 1988, pp. 30 - 37. New York: ACM Press.

Fox, E. A. (1991). Advances in Interactive Digital Multimedia Systems. *IEEE Computer* 24(10), October 1991, pp. 9 - 21.

Goodman, G. and Abel, M. (1986) Collaboration Research in SCL. *Proceedings of CSCW'86 Conference on Computer Supported Cooperative Work* , Austin, TX, December 3 - 5, 1990, pp. 246 - 252, New York: ACM Press.

Greenberg, S. (1990). Sharing Views and Interactions with Single-User Applications. *Proceedings of COIS'91 Conference on Office Information Systems,* Cambridge, MA, April 25-27, 1990, pp. 227-237. New York: ACM Press.

Greenberg, S. and Bohnet, R. (1991). GroupSketch: A Multi-User Sketchpad for Geographically Distributed Small Groups. *Proceedings of Graphics Interface'91*, Calgary, Alberta, June 5 - 7, 1991.

Group Technologies (1991). *Aspects: The First Simultaneous Conference Software for the Macintosh, Version 1.* Manual, Group Technologies, Inc., Arlington, VA.

Hayne, S. and Ram, S. (1990) Multi-user View Integration System: An Expert System for View Integration. *Proceedings of the IEEE International Conference on Data Engineering*, Los Angeles, CA, pp. 402-409.

Kraut, R., Egido, C. and Galegher, J. (1988). Patterns of Contact and Communication in Scientific Research Collaboration. *Proceedings of the Conference on Computer-Supported Cooperative Work*, Portland, OR, September 26-28, 1988, pp. 1 - 12. New York: ACM Press.

Lantz, K. A. (1986). An Experiment in Integrated Multimedia Conferencing. *Proceedings of CSCW'86 Conference on Computer-Supported Cooperative Work*, Austin, TX, December 1986, pp. 267 - 275. New York: ACM Press.

Lauwers, J. C. and Lantz, K. A. (1990). Collaboration Awareness in Support of Collaboration Transparency: Requirements for the Next Generation of Shared Window Systems. *Proceedings of CHI'90 Conference on Human Factors in Computing Systems,* Seattle, WA, April 1990, pp. 303 - 312. New York: ACM Press.

Lauwers, J. C., Joseph, T. A., Lantz, K. A. and Romanow, A. L. (1990). Replicated Architectures for Shared Window Systems: A Critique. *Proceedings of COIS'90 Conference on Office Information Systems*, Cambridge, MA, April 25-27, 1990, pp. 249 - 260. New York: ACM Press.

Li, J. and Mantei, M. (1992). Working Together, Virtually. *Proceedings of Graphics Interface, 92*, Vancouver, BC, Canada, May 13-15, 1992.

Lu, I. M. (1992). *Supporting Idea Management in a Shared Drawing Tool.* Unpublished Masters dissertation, Department of Computer Science, University of Toronto, Toronto, ON, Canada, January 1992.

Lu, I. and Mantei, M. (1992). *Managing Design Ideas with a Shared Drawing Tool.* Unpublished Manuscript, Department of Computer Science, University of Toronto, January, 1992.

Mantei, M. M., Baecker, R. M., Sellen, A.J., Buxton, W. A., Milligan, T. and Wellman, B. (1991). Experiences in the Use of a Media Space. *Proceedings of CHI'91 Donference on Human Factors in Computing Systems,* New Orleans, LA, April 28 - May 4, 1991, pp. 203 - 208. New York: ACM Press.

Mantei, M. (1988). Capturing the Capture Lab Concepts: A Case Study in the Design of Computer Supported Meeting Environments. *Proceedings of CSCW'88 Conference on Computer Supported Cooperative Work*, Portland, OR, September 26-28, 1988, pp. 257 - 270. New York: ACM Press.

Mawby, K. L. (1991). *Designing Collaborative Writing Tools.* Unpublished Masters dissertation, Department of Computer Science, University of Toronto, Toronto, ON, Canada, September 1991.

Neuwirth, C. M., Kaufer, D. S., Chandhok, R. and Morris, J. H. (1990). Issues in the Design of Computer Support for Co-authoring and Commenting. *Proceedings of CSCW'90 Conference on Computer Supported Cooperative Work* , Los Angeles, CA, October 7 - 10, 1990, pp. 183 - 195. New York: ACM Press.

Patterson, J. F., Hill, R. D., Rohall, S. L. and Meeks, W. S. (1990). Rendezvous: An Architecture for Synchronous Multi-user Applications. *Proceedings of CSCW'90 Conference on Computer Supported Cooperative Work* , Los Angeles, CA, October 7 - 10, 1990, pp. 317 - 328. New York: ACM Press.

Posner, I. R. and Baecker, R. M. (1991). How People Write Together. *Proceedings of the 25th Annual Hawaii International Conference on Systems Science, Vol IV* Kauai, HI, January 7 - 10, 1992, pp. 127 - 138.

Root, R. W. Design of a Multi-Media Vehicle for Social Browsing. *Proceedings of CSCW'88 Conference on Computer Supported Cooperative Work* , Portland, OR, September 26 - 28, 1988, pp. 25 - 38. New York: ACM Press.

Scheifler, R. W., Gettys, J. and Newman, R. (1988) *X Window System*. Burlington, MA: Digital Press.

Stefik, M., Foster, G., Bobrow, D. G., Kahn, K., Lanning, S. and Suchman, L. (1987). Beyond the Chalkboard: Computer Support for Collaboration and Problem Solving in Meetings. *Communications of the ACM* 30(1), January 1987, pp 32 - 47.

Tang, J. C. (1989) Listing, Drawing and Gesturing in Design: A Study of the Use of Shared Workspaces by Design Teams. Unpublished Ph.D. Dissertation, Stanford University, April 1989. Also available as*Xerox Technical Report SSL-89-3*., Xerox PARC, Palo Alto, CA.

Tang, J. C. and Minneman, S. L. (1990). VideoDraw: A Video Interface for Collaborative Drawing. *Proceedings of CHI'90 Conference on Human Factors in Computing Systems*, Seattle, WA, April 1990, pp. 313 - 320. New York: ACM Press.

University of Michigan. (1990) ShrEdit 1.0: A Shared Editor for the Apple Macintosh -- User's Guide and Technical Description, June 1990. Cognitive Science and Machine Intelligence Laboratory, University of Michigan, Ann Arbor, MI.

Vin, H. M., Zellweger, P. T., Swinehart, D. C. and Rangan, P.V. (1991). Multimedia Conferencing in the Etherphone Environment. *IEEE Computer* 24(10), October 1991, pp. 69 - 79.

Watabe, K., Sakata, S., Maeno, K., Fukuoka, H. and Ohmori, T. (1990). Distributed Multiparty Desktop Conferencing System: MERMAID. *Proceedings of CSCW'90 Conference on Computer Supported Cooperative Work* , Los Angeles, CA, October 7 - 10, 1990, pp. 27 - 38. New York: ACM Press.

Movement Time Prediction in Human-Computer Interfaces

I. Scott MacKenzie
Department of Computing and Information Science
University of Guelph
Guelph, Ontario, Canada N1G 2W1
519-824-4120, mac@snowhite.cis.uoguelph.ca

Abstract

The prediction of movement time in human-computer interfaces as undertaken using Fitts' law is reviewed. Techniques for model building are summarized and three refinements to improve the theoretical and empirical accuracy of the law are presented. Refinements include (1) the Shannon formulation for the index of task difficulty, (2) new interpretations of "target width" for two- and three-dimensional tasks, and (3) a technique for normalizing error rates across experimental factors. Finally, a detailed application example is developed showing the potential of Fitts' law to predict and compare the performance of user interfaces before designs are finalized.

Keywords: human performance modeling, Fitts' law, input devices, input tasks

Introduction

Movement is ubiquitous in human-computer interaction. Our arms, wrists, and fingers busy themselves on the keyboard and desktop; our head, neck, and eyes move about attending to graphic details recording our progress. Matching the movement limits and capabilities of humans with interaction techniques on computing systems, therefore, can benefit from research in this important dimension of human behaviour.

One focus in HCI research is in predicting and modelling the time for humans to execute tasks. Although encompassing a vast territory to be sure, one dimension is the time invested in movement. In fact, movement is an integral, seemingly innocuous component of many research questions in HCI: Are popup menus superior to menu bars? Which input device affords the quickest and most accurate interaction? Should a scroll bar in a text editor be on the left or right side of the CRT display? Can gestures replace commands in text editing?

In this review paper, we will explore Fitts' law, a powerful model for the prediction of movement time in human-computer interaction. We are motivated by (a) apparent problems in previous work, (b) the difficulty in interpreting and comparing published results, and (c) the need to guide future research using Fitts' law.

We begin with a brief tour of Fitts' law, and follow by describing some refinements to correct flaws or to improve its prediction power. Finally, derived models are used in an application example to illustrate the potential of Fitts' law in assessing and comparing interface scenarios before they are finalized in products.

A Brief Tour of Fitts' Law

The application of information theory to human performance modeling dates to the 1950s when experimental psychologists (e.g., Miller, 1953) embraced the work of Shannon, Wiener, and other information theorists as a framework for understanding human perceptual, cognitive, and motor processes. Models, or "laws", that persist today include the Hick-Hyman law for choice reaction time (Hick, 1952; Hyman, 1953) and Fitts' law for movement time (Fitts, 1954; Fitts & Peterson, 1964).

According to Fitts, a movement tasks' difficulty (*ID*, the "index of difficulty") can be quantified using information theory by the metric "bits". Specifically,

$$ID = \log_2(2A / W) \qquad (1)$$

where *A* is the distance or amplitude to move and *W* is the width or tolerance of the region within which the move terminates. Because *A* and *W* are both measures of distance, the term within the parentheses in Equation

1 is without units. The unit "bits" emerges from the somewhat arbitrary choice of base 2 for the logarithm. From Equation 1, the time to complete a movement task is predicted using a simple linear equation, where movement time (*MT*) is a linear function of *ID*.

Figure 1 shows the serial tapping task used by Fitts (1954). In this experiment, subjects alternately tapped as quickly and accurately as possible between two targets of width *W* at a distance *A*. Obviously, as targets get farther away or as they get smaller, the tasks get more difficult and more time is required to complete the task. In Fitts' experiment *A* and *W* each varied over four levels. The easiest task had $A = 1$ inch and $W = 1$ inch for $ID = \log_2(2A/W) = \log_2(2) = 1$ bit. The hardest task had $A = 16$ inches and $W = 0.25$ inches for $ID = \log_2(128) = 7$ bits.

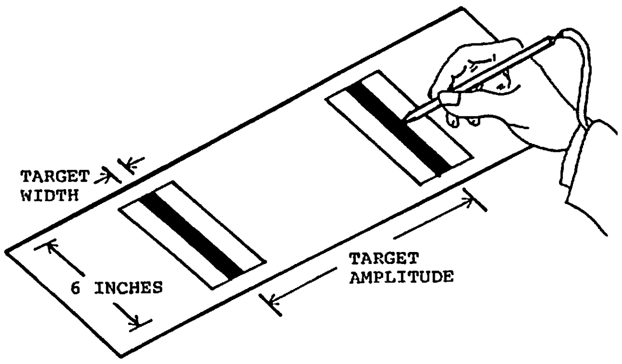


Figure 1. The serial tapping task used by Fitts (1954).

The task in Figure 1 can be implemented on interactive graphics systems using targets displayed on a CRT and a cursor manipulated by an input device. A common variation is the discrete task — a single movement toward a target from a home position (see Figure 2). Target selection is usually accomplished by a button push when the cursor is over the target.

The first use of Fitts' law in HCI research was the work of Card, English, and Burr (1978) who applied the model on data collected in a text selection task using a joystick and a mouse. Subjects were required to move the cursor from a home position to a target — a word — and select it by pushing a button. Numerous other HCI researchers have subsequently used Fitts' law. Examples include Boritz, Booth, and Cowan (1991); Gillan, Holden, Adam, Rudisill, and Magee (1990); Card, Mackinlay, and Robertson (1990); Epps (1986); MacKenzie, Sellen, and Buxton (1991); Walker and Smelcer (1990); and Ware and Mikaelian (1987).

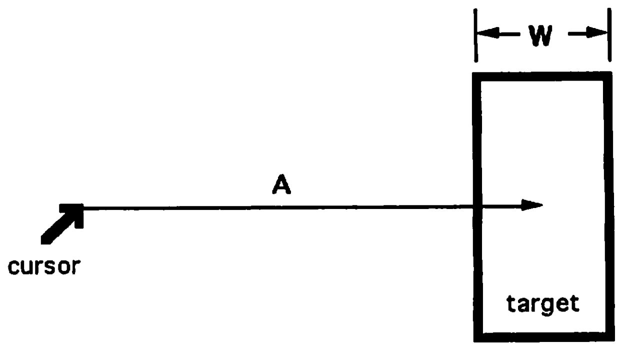


Figure 2. A discrete task using a cursor and a target displayed on a CRT.

A movement model based on Fitts' law is an equation predicting movement time (*MT*) from a task's index of difficulty (*ID*). Figure 3 shows the general idea.

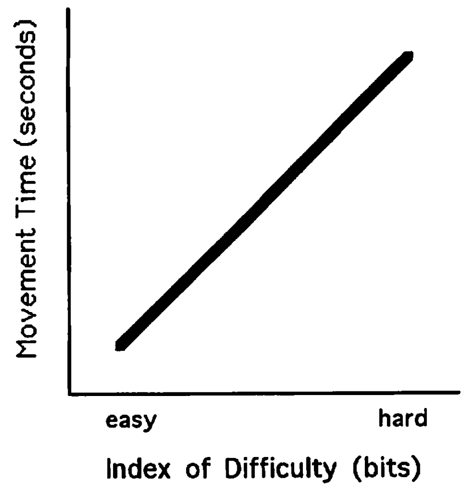


Figure 3. Movement time prediction.

As expected, movement time for hard tasks is longer than for easy tasks. The prediction equation for the line in Figure 3 is of the form

$$MT = b\,ID \quad (2)$$

or

$$MT = a + b\,ID \quad (3)$$

depending on whether or not the line goes through the origin. In both cases, b is the slope of the line.

Since task difficulty is analogous to information, the rate of task execution can be interpreted as the human rate of information processing. For example, if a task rated at, say, $ID = 4$ bits is executed in 2 seconds, then the human rate of information processing is $4/2 = 2$ bits/s. This is also evident by examining Figure 3. Since the vertical and horizontal axes carry the units "seconds" and "bits" respectively, the slope of the line is in "seconds/bit". The reciprocal of the slope is in "bits/s". The latter measure Fitts called the index of performance (*IP*). Since *IP* is in bits/s, the term "bandwidth" is also used.

Intuitively, the higher the bandwidth the higher the rate of human performance since more information is being articulated per unit time. One of the strengths in Fitts' law is that measures for *IP*, or bandwidth, can motivate performance comparisons across factors such as device, limb, or task. It follows that performance in a human-computer interface can be optimized by selecting and combining those conditions yielding high bandwidths.

Equation 2 is ideal since the prediction line goes through the origin. This is important for the intuitive reason that a movement task rated at $ID = 0$ bits is predicted by Equation 2 to take zero seconds, as desired. By Equation 3, however, a non-zero intercept implies that a task rated at $ID = 0$ bits will take "a" seconds to execute. (This point will surface again later.)

Building a Fitts' Law Model

In building a Fitts' law model, the slope and intercept coefficients in the prediction equation are determined through empirical tests. The tests are undertaken in a controlled experiment using a group of subjects and one or more input devices and task conditions. On interactive computing systems, this could range from manipulating a cursor with a mouse and selecting icons to manipulating a virtual hand with an input glove and grabbing objects in a 3D virtual space.

The design of experiments for Fitts' law studies is simple. Tasks are devised to cover a range of difficulties by varying *A* and *W*. For each task condition, multiple trials are conducted and the time to execute each is recorded and stored electronically for statistical analysis. Errors are also recorded (and analysed as discussed later). Generally, measurements are aggregated across subjects resulting in one data point for each task condition. A typical data set is shown in Table 1 for a stylus in a serial point-select task mimicking Fitts' serial tapping task (MacKenzie, 1991).

Table 1
Data From an Experiment Using a Stylus in a Point-Select Task

A[a]	*W*[a]	*ID* (bits)	*MT* (ms)	Errors (%)	*IP* (bits/s)
8	8	1	254	0.0	4.3
8	4	2	353	1.9	6.1
16	8	2	344	0.8	6.4
8	2	3	481	1.7	6.4
16	4	3	472	2.1	6.6
32	8	3	501	0.6	6.2
8	1	4	649	8.8	6.3
16	2	4	603	2.1	6.8
32	4	4	605	2.7	6.7
64	8	4	694	2.5	5.9
16	1	5	778	7.0	6.6
32	2	5	763	3.4	6.6
64	4	5	804	2.3	6.3
32	1	6	921	8.5	6.6
64	2	6	963	3.3	6.3
64	1	7	1137	9.9	6.3
		Mean	645	3.6	6.3
		SD	243	3.1	0.6

[a]experimental units; 1 unit = 8 pixels

The first three columns contain the independent variables target amplitude (*A*), target width (*W*), and the associated index of difficulty (*ID*) calculated using Equation 1. *A* and *W* each varied over four levels, yielding *ID*s of 1 to 7 bits. In the last three columns are the dependent variables movement time (*MT*), error rate, and the index of performance ($IP = ID/MT$). Each row entry is the mean of about 470 trials. The grand means were 645 ms for movement time, 3.6% for error rate, and 6.3 bits/s for the index of performance, as shown in the second last row.

The next step in model building is to enter the 16 *MT*-*ID* points in tests of correlation and linear regression. The data in Table 1 yield a regression line with movement time (ms) predicted as

$$MT = 53 + 148\, ID \tag{4}$$

with a correlation of $r = .992$. Correlations above .900 are considered very high for any experiment involving measurements on human subjects. A high r suggests that the model provides a good description of observed behaviour.

The prediction equation has an intercept of 53 ms and a slope of 148 ms/bit. Converting the slope to its reciprocal yields an index of performance, or bandwidth, of 6.8 bits/s. Often the data points and regression line are shown together in a scatter plot (see Figure 4).

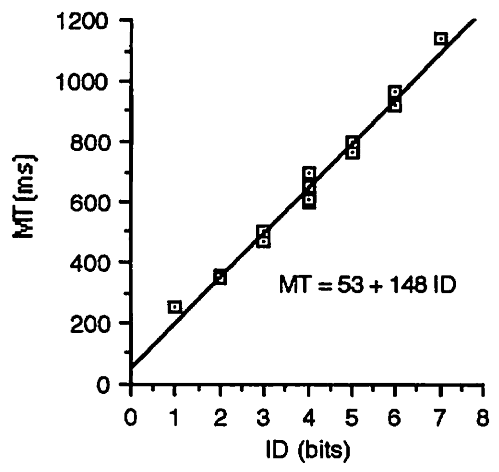


Figure 4. Scatter plot and regression line for the data in Table 1.

As just shown, the index of performance can be calculated using a direct division of mean scores ($IP = ID/MT$) or through linear regression ($IP = 1/b$, from $MT = a + b\,ID$). The two methods yielded slightly different results: 6.3 bits/s using the direct method vs. 6.8 bits/s using the slope reciprocal. Although the disparity seems small, the correct method of calculation is in doubt. Fitts used the direct method in his 1954 paper and linear regression in a subsequent study (Fitts & Peterson, 1964). Most (but not all) current researchers use linear regression. Notably, the disparity is systematic: If the line in Figure 4 is rotated counter-clockwise forcing it through the origin, the slope increases and the slope reciprocal decreases, thus reducing the disparity.

If the regression line intercept is small the difference between the two bandwidth measures will be slight. If the intercept is large, the difference may be appreciable. In the latter case, a consistent, additive component of the task (such as a button push) may be the source of the intercept.

An example is the model built by Card et al. (1978) for the mouse in a text selection task:

$$MT = 1030 + 96\,ID \tag{5}$$

Although the fit was again very good ($r = .91$), this equation has one of the largest intercepts in published Fitts' law research. The implication, of course, is that a movement task rated at $ID = 0$ bits will take 1.03 seconds.

The large intercept also casts doubt on the validity of the slope reciprocal as a measure for bandwidth. The slope of 96 ms/bit translates into a bandwidth of 10.4 bits/s. The Card et al. (1978) experiment used 20 task conditions with a mean ID of 2.63 bits/s. The mean movement time was reported as 1.29 s; so, the bandwidth calculated using a direct division of means is 2.63/1.29 or 2.0 bits/s. This differs from the slope reciprocal bandwidth by a factor of five! This is a concern if one wishes to generalize findings in terms of the human rate of information processing. Is the rate in this case 10.4 bits/s or 2.0 bits/s? As evident in Table 2, the regression line equations (and resulting bandwidths) vary tremendously in previous Fitts' law research using the mouse in point-select tasks.

A third possibility for calculating the prediction model is regression-through-the-origin. Although it has never been employed, the method will produce the best-fitting line passing through the origin.

Since linear regression produces the prediction line with the best fit, it is the preferred choice for model building. However, the proviso is added that the intercept must be small. "Small" in this context is on the order of a few hundred milliseconds — a value which can reasonably be attributed to random variation in measurements.

Table 2
Prediction Equations and Bandwidths From Fitts' Law Studies Using a Mouse in Point-Select Tasks

Study	Prediction equation (ms)	Bandwidth (bits/s)
Boritz et al., 1991	$MT = 1320 + 430\,ID$	2.3
Epps, 1986	$MT = 108 + 392\,ID$	2.6
MacKenzie et al., 1991	$MT = -107 + 223\,ID$	4.5
Han et al., 1990	$MT = 389 + 175\,ID$	5.7
Card et al., 1978	$MT = 1030 + 96\,ID$	10.4
Gillan et al., 1990	$MT = 795 + 83\,ID$	12.0

Refinements to Fitts' Law

Despite an extremely good fit in empirical tests, Fitts' law is a frequent target for critical reviews (e.g., Meyer, Smith, Kornblum, Abrams, & Wright, 1990; MacKenzie, in press; Welford, 1968). Numerous problems surface under close examination or when findings are compared across studies. These have motivated corrections or refinements to the model. In this section we review these and offer suggestions to guide researchers in applying the law.

Formulation for Index of Difficulty

Early examinations of the law noted a consistent departure of data points above the regression line for "easy" tasks ($ID < 3$ bits). A new formulation for ID was proposed by Welford (1960) to correct this:

$$ID = \log_2(A/W + 0.5). \qquad (6)$$

Many researchers, including Fitts, noted an improved fit using Equation 6. The Welford formulation was used by Card et al. (1978), whose findings were subsequently elaborated in the *Psychology of human-computer interaction* (Card, Moran, & Newell, 1983). Not surprisingly, many HCI researchers citing Card and colleagues adopt the Welford formulation (e.g., Boritz et al., 1991; Gillan et al., 1990).

It has also been argued that Fitts, in formulating his model, deviated unnecessarily from Shannon's original work in information theory (MacKenzie, 1989; Shannon & Weaver, 1949), and that a more theoretically sound formulation for the index of task difficulty is

$$ID = \log_2(A/W + 1). \qquad (7)$$

In terms of MT, the prediction model becomes

$$MT = a + b \log_2(A/W + 1). \qquad (8)$$

Equation 8, known as the Shannon formulation, is preferred because it

- provides a slightly better fit with observations,
- exactly mimics the information theorem underlying Fitts' law, and
- always gives a positive rating for the index of task difficulty.

The last point above is understood by examining the three formulations for ID. Using the Fitts or Welford formulation (Equations 1 & 6), the index is negative if the amplitude is less than half the target width; that is, $A < W/2$. At the very least, a negative rating for task difficulty is a nuisance. From Equation 7, as A approaches zero (for any W), ID approaches 0 bits, but never becomes negative. Obviously, the latter effect has strong intuitive appeal. Although for the one-dimensional paradigm an amplitude less than $W/2$ only occurs when the starting position is inside the target (see Figure 2), very small $A{:}W$ ratios are fully possible when the law is applied in two dimensional tasks. This is demonstrated in the next section.

Extension to Two Dimensions

It is important to remember that the experiments undertaken by Fitts and most other experimental psychologists tested one dimensional movements. This is evident in Figures 1 and 2. Since both target amplitude and target width are measured along the same axis, it follows that the model is inherently one dimensional.

HCI researchers employing Fitts' law invariably use target selection tasks realized on a two-dimensional CRT display. The shape of the target and the angle of approach, therefore, must be considered carefully in applying the model. If the targets are circles (or perhaps squares), then the one-dimensional constraint remains largely intact. (The "width" of a circle is the same, regardless of the angle of measurement!) However, if targets are rectangles (e.g., "words"), the situation is confounded. We can still view the amplitude as the distance to the centre of the target; but the definition of target width is unclear. This is illustrated in Figure 5.

In applying the model in 2D tasks, a critical question arises: What is target width? The default strategy is to consistently use the horizontal extent of the target. We call this the "STATUS QUO" model for target width. Unfortunately, a STATUS QUO model yields unrealistically low (sometimes negative!) estimates for task difficulty when, for example, a short and wide target, such as a word or series of words, is approached from above or below at close range. At least two examples of this exist in the literature. Gillan et al. (1990) used Fitts' law in a target selection task using strings of characters as targets while varying the approach angle and approach distance. One extreme condition saw a 26-character (6 cm) target approached diagonally from a distance of 2 cm. Welford's formulation was used, so the index of task difficulty was $ID = \log_2(A/W + 0.5) = \log_2(2/6 + 0.5) = -2.6$ bits. The negative rating is troublesome. (A similar example is found in Card et al., 1978).

One result of task difficulty extending to the left of $ID = 0$ bits is that it becomes certain that a positive (probably large) intercept emerges under linear regression. This is because conditions with $ID = 0$ bits or less correspond to conditions that *actually occurred* in the experiment.

No doubt, such tasks will take a non-trivial (positive) amount of time.

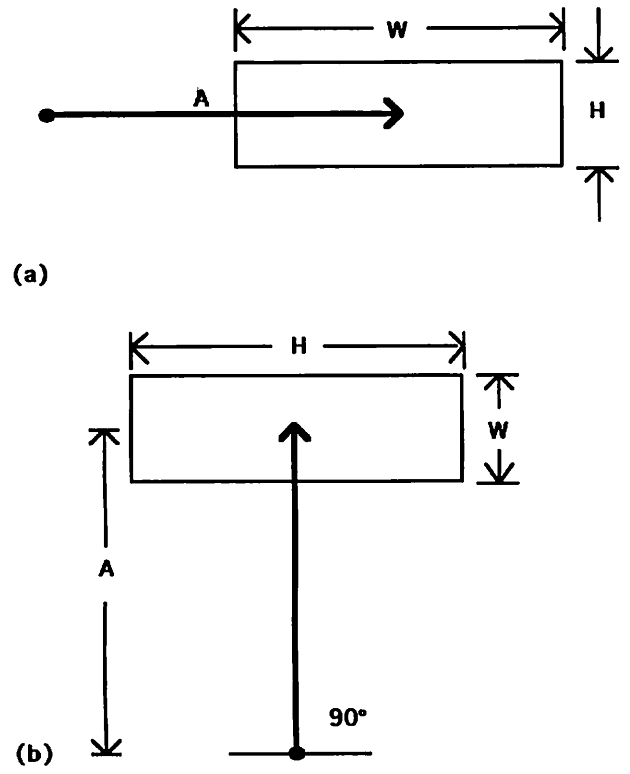


Figure 5. Fitts' law in two dimensions. The roles of width and height reverse as the approach angle changes from 0° to 90°.

We suggest two ways to correct this problem. The first is to use the Shannon formulation for *ID*, which for the example above would increase the rating to $\log_2(2/6 + 1) = +0.42$ bits.

A second and additional strategy is to substitute for *W* a measure more consistent with the 2D nature of the task. Consider Figure 6. The inherent 1D constraint in the model is maintained by measuring *W* along the approach axis. This is shown as *W'* (read "W prime") in the figure. Notwithstanding the assertion that subjects may "cut corners" to minimize distances, the *W'* model is appealing because it allows a 1D interpretation of a 2D task.

Another possible substitution for target width is "the smaller of W or H". This pragmatic approach has intuitive appeal in that the smaller of the two dimensions seems more indicative of the accuracy demands of the task. We call this the "SMALLER-OF" model. This model is computationally simple since it can be applied only knowing *A*, *W*, and *H*. The *W'* model, on the other hand, requires *A*, *W*, *H*, and θ_A, and a geometric calculation to determine the correct substitution for *W*. The SMALLER-OF model is limited to rectangular targets, however.

An experiment was conducted to test the different models for target width on a standard target selection task using a mouse. The design employed a balanced range of short-and-wide and tall-and-narrow targets approached from various angles. The results indicate that both the SMALLER-OF and *W'* models are empirically superior to the STATUS QUO model and that the difference between the SMALLER-OF and *W'* models is insignificant (MacKenzie & Buxton, in press).

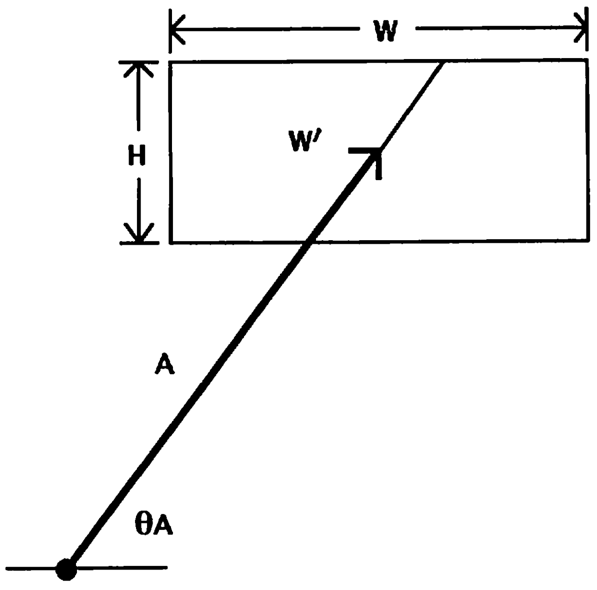


Figure 6. What is target width? Possibilities include *W'* (the width of the target along an approach vector) or the smaller of *W* or *H*.

In summary, the SMALLER-OF or *W'* model is recommended in applying Fitts' law to two dimensional tasks. We should note that extensions to three dimensional tasks easily follow from the arguments above; although Fitts' law has yet to be tested in 3-space. Finally, note that the *W'* model reduces to the STATUS QUO model for one dimensional tasks.

Normalization and the Speed-Accuracy Tradeoff

The reciprocity between the speed of actions and the subsequent accuracy of responses has been well documented in experimental psychology and human factors engineering (e.g., Hancock & Newell, 1985;

Pew, 1969). Despite this, researchers all too often base performance analyses solely or largely on task completion time measurements while paying little regard to the errors that accompanied — and equally contributed to — performance. Comparative evaluations based on movement time criteria are difficult when faced with disparities in error rates. That is, if condition A was faster than condition B, but had more errors, it is uncertain which condition is better.

The most important refinement to Fitts' law, perhaps, is the technique for accommodating spatial variability or errors in responses. The techniques calls for target width to be adjusted based on the distribution of "hits" (selection coordinates) about each target. Thus, at the model building stage, *W* is a dependent variable rather than an independent variable. The claim is that the technique increases the power of Fitts' law since normalized models inherit a known and consistent error rate. In particular, comparisons within and between studies are strengthened by a "level playing field".

The output or "effective" target width (W_e) is derived from the observed distribution of "hits", as described by Crossman and Welford (Welford, 1968, p. 147). This adjustment lies at the very heart of the information-theoretic metaphor — that movement amplitudes are analogous to "signals" and that end-point variability (viz., target width) is analogous to "noise".

The technique is illustrated in Figure 7. When a nominal error rate of 4% occurs (Figure 7a), no adjustment is required ($W_e = W$). When a different error rate occurs, target width is adjusted by multiplying it by a ratio of *z* scores obtained from statistical tables for the unit-normal curve. For example, if 2% errors were recorded on a block of trials when selecting a 5 cm wide target, then $W_e = 2.066/2.326 \times 5 = 4.45$ cm (Figure 7b).[1] The analyses proceed as before except using "effective *IDs*", calculated using W_e instead of *W*.

Applying this technique is essential if comparisons within or across studies are attempted. Rephrasing and earlier point, if performance on condition A was 6 bits/s and performance on condition B was 5 bits/s, we'd like to conclude that condition A was superior; however, in the absence of identical or normalized error rates, such a conclusion is weak and perhaps wrong. Fitts' law models have appeared in published research accompanied by error rates from 0% to 25% (see MacKenzie, in press); yet the technique is rarely applied. It is strongly recommended that future research adopt the method.

[1]The technique is described elsewhere in full detail with examples (MacKenzie, in press).

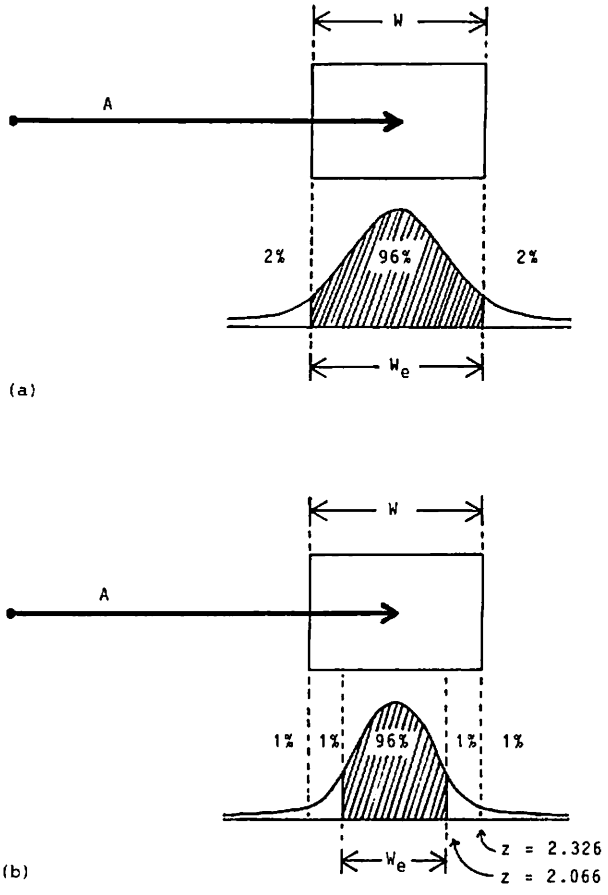


Figure 7. The method for normalizing responses. In (a) the error rate is 4% so no adjustment is needed. In (b) the rate is 2% so an adjustment is made.

Applying Fitts' Law: An Example

One challenge in research is relating the findings to real-world problems confronting practitioners in the field. Despite its recognition as one of the more robust models for human movement, Fitts' law has rarely migrated from the research lab.

An instance of Fitts' law actually being used in a product is a computer-aided design tool called *Jack*, developed at the University of Pennsylvania (Badler, Webber, & Kalita, 1991). In *Jack* three dimensional human figures (simulated on a CRT display) are programmed to execute motions such a picking up objects or making adjustments on a simulated control panel. Since the researchers did not know how fast to program movements in *Jack*, they called upon Fitts' law. A derived Fitts' law model was embedded in *Jack* to provide the duration for movements, based on the distance to move and the size of the terminating region for the move. The visual result is quite natural.

In this section, further applications for Fitts' law models are explored and a specific example is developed.

First, we must recognize when *not* to use Fitts' law. The law is a prediction model for rapid, aimed movement. A variety of user input activities do not fit this description, including drawing, inking, writing cursive script, and other temporarily constrained tasks. Furthermore, some devices are inadequately modeled by Fitts' law. Isometric joysticks are force sensing and undergo negligible motion. As a model for human movement, it seems odd to apply Fitts' law where no limb movement takes place. Card et al. (1978) found that performance data for an isometric joystick were poorly described by Fitts' law. After the data were decomposed by amplitude, however, the model fit well, yielding a series of parallel lines across amplitude conditions. Other devices, such as those for velocity control, may also display characteristics inappropriate for a Fitts' law model.

For the more common pointing devices (such as the mouse, trackball, or stylus), and for common point-select or drag-select tasks, however, Fitts' law has the potential to assist in the design and evaluation of graphical user interfaces. Questions of the form "How long will this task take?" can be answered using Fitts' law prediction equations if certain conditions exist. If the tasks are rapidly executed with negligible or known mental preparation time, system response time, device homing time, etc., then there is a good chance Fitts' law can assist in evaluating alternative methods.

Consider the case of deleting a file (icon) on the Apple *Macintosh* computer. Three possible methods are listed below.

DRAG-SELECT: Drag the icon to the trash-can. (This is the traditional *Macintosh* method.)

POINT-SELECT: Select the file icon with a point-select operation, then select the trash-can icon with a second point-select operation.

STROKE-THROUGH: Stroke through the icon. This method uses a button-down action beside the file icon followed by dragging (stroking) through the icon and a button-up action on the opposite side.

The STROKE-THROUGH method is an example of input which mimics a natural gesture in manuscript editing (Hardock, 1991; Kurtenbach & Buxton, 1991). A reasonable assumption for the stroking gesture is that the button-down action occurs on the left of the icon in the centre of a region the same size as the icon, and that the button-up action occurs similarly in a region on the right of the icon.

A possible screen layout is shown in Figure 8. The trashcan icon is located at the bottom right of the screen. The file icon is placed in the middle of the screen at a distance of 14 cm from the trashcan. Both icons are 2 cm square. The button-down and button-up regions for the stroking gesture are shown in dotted lines.

The calculations proceed using derived pointing and dragging models for the mouse (MacKenzie et al., 1991):

Pointing model:

$$MT = 230 + 166\, ID \quad (IP = 6.0 \text{ bits/s}) \tag{9}$$

Dragging model:

$$MT = 135 + 249\, ID \quad (IP = 4.0 \text{ bits/s}) \tag{10}$$

The pointing model applies to the POINT-SELECT method, while the dragging model applies to the DRAG-SELECT and STROKE-THROUGH methods. Note also that for the STROKE-THROUGH method the amplitude is 4 cm, rather than 14 cm.

Once the initial move to the icon is complete, the time to delete the file icon using each method is calculated as follows:

DRAG-SELECT:

$$\begin{aligned} MT &= 135 + 249 \log_2(14/2 + 1) \\ &= 135 + 249 \times 3 \\ &= 882 \text{ ms} \end{aligned} \tag{11}$$

POINT-SELECT:

$$\begin{aligned} MT &= 230 + 166 \log_2(14/2 + 1) \\ &= 230 + 166 \times 3 \\ &= 728 \text{ ms} \end{aligned} \tag{12}$$

STROKE-THROUGH:

$$\begin{aligned} MT &= 135 + 249 \log_2(4/2 + 1) \\ &= 135 + 249 \times 1.58 \\ &= 528 \text{ ms} \end{aligned} \tag{13}$$

This result suggests, using Fitts' law analyses, that the traditional method of deleting a file on the *Macintosh* is slower than two alternate methods. (The STROKE-THROUGH method is 40% faster.) The example is simplistic, however. Other issues such as methods for deleting multiple files or for un-deleting files must be considered too.

Even though the rate of information processing is lower during dragging than during pointing, the STROKE-THROUGH method, which is a dragging operation, is faster than the POINT-SELECT method. This is due to the combined effect of the intercepts in the prediction equations and the different movement amplitudes. Using the stroke-through method, the predicted *MT* is independent of the distance (A) between the file icon and the trashcan icon since the movement amplitude is nominally set at twice the file icon's width. However, the predicted *MT* decreases with A for the POINT-SELECT method. This suggests there may be a cross-over point below which the POINT-SELECT method is faster. Knowledge of this may play a critical role in selecting an appropriate method. In the current example, the cross-over point is calculated by equating the POINT-SELECT and STROKE-THROUGH predictions as follows:

$$230 + 166 \log_2(A/2 + 1) = 135 + 249 \log_2(A/2 + 1) \qquad (14)$$

Solving Equation 14 yields $A = 2.72$ cm. So, the point-select method is faster than the STROKE-THROUGH method only for amplitudes less than 2.72 cm. Certainly, this is a minority of cases. Other possibilities could be explored too, such as increasing the size of the trashcan icon relative to file icons.

Future Possibilities

In the hard science-soft science debate, Newell and Card (1985) hold that "striving to develop a theory that does task analysis by calculation is the key to hardening the science" (p. 237). Indeed, future applications of Fitts' law may include "embedded models" as an

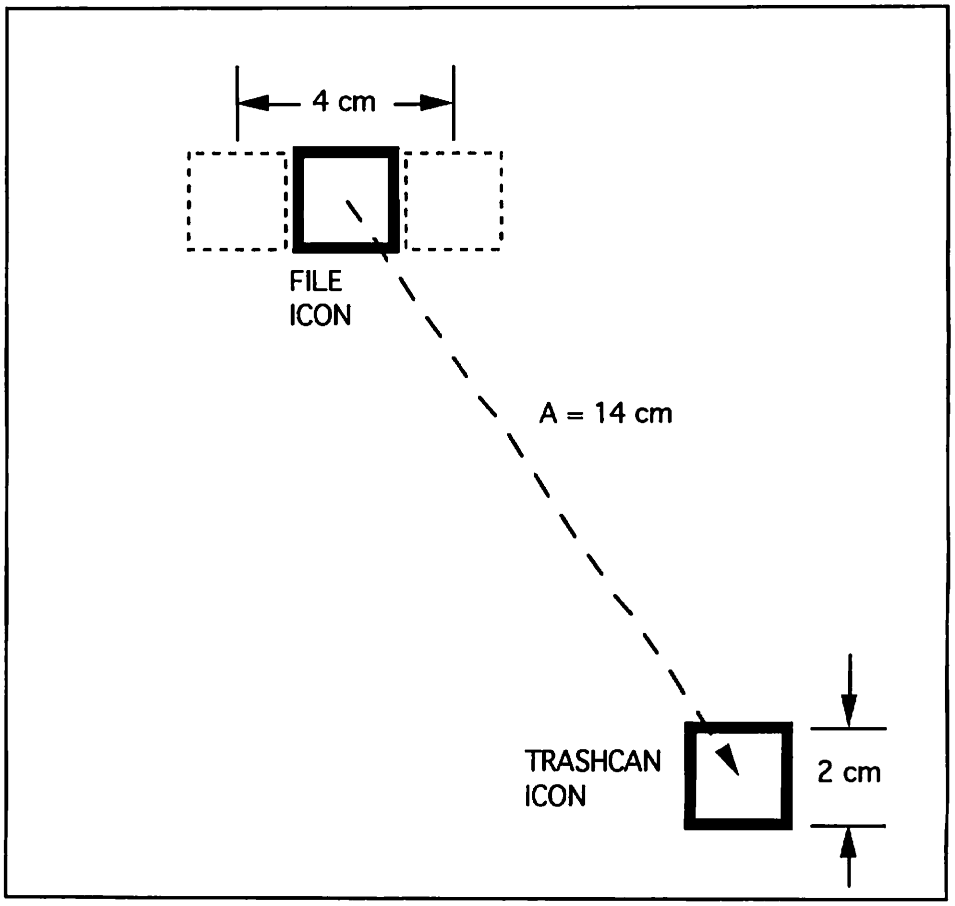


Figure 8. Screen layout for Fitts' law example.

integral part of user interface design and management systems. The scenario envisioned goes something like this: A user interface is patched together in story-board fashion with a series of screens (with their associated soft buttons, pull-down menus, icons, etc.) and interconnecting links. The designer puts the embedded model into "analyse mode" and "works" the interface — positioning, drawing, selecting, and "doing" a series of typical operations. When finished, the embedded model furnishes a predicted minimum performance time for the tasks (coincident with a nominal or programmable error rate). The designer moves, changes, or scales objects and screens and requests a reanalysis of the same task.

This is a rough first approximation. At a higher level, an embedded model is an "agent" in beta testing, monitoring activities and profiling performance. The profile catalogs a variety of facets of the interface. For example, a "locus of control" could identify frequently used screens or high-use regions on a display. In a word processing system, for example, depending on the implementation for scrolling, a user may work mostly at the bottom of the screen. Knowledge of this may imply that a menu bar could be placed at the bottom of the display rather than at the top.

A more powerful embedded model performs sequence analysis. How long does it take to get from point A to point B through a series of intermediate steps? Or, of alternate ways, which is the fastest? Hypermedia environments with embedded links facilitate such analyses, since an explicit and external state-transition description may not be needed. The agent acts on the screen definitions and links (as they exist in the application) in evaluating alternative or optimal paths.

An embedded model is more than a software routine incorporating Fitts' law. System and user performance constants are needed, similar to those in the Keystroke-Level Model (Card, Moran, & Newell, 1980). A parametric analysis could identify bottlenecks or optimal combinations. For example, decreasing pointing time by 10% vs. decreasing user keystroke time by 10% may have vastly different effects on overall task completion time. If a task can be accomplished two ways (e.g., 4 point-select operations vs. 20 keystrokes), which is the fastest? A parametric analysis could identify cross-over points across settings (as shown earlier). The designer could establish ranges and weights for parameters, and an agent, armed with embedded models, would take it from there.

Fitts' law may also participate in user-adaptive systems — systems with a human interface which changes to accommodate a user's capabilities and limitations (Rouse, 1988). Control systems for air traffic, ground traffic, power generation or industrial processes are potential instances. One can imagine several human operators interacting with a complex system by manipulating iconic controls in response to system events. As system dynamics change, the demands on operators change. Models such as Fitts' law (and/or the Hick-Hyman law) could measure the load on operators (in bits/s) or predict their performance. In safety-critical settings, it may be possible to systematically allocate tasks to workers to maintain set-points of sub-maximal performance.

Human-computer interaction has advanced by leaps and bounds in recent years. We can attribute this primarily to the improved interfaces advanced through the technologies of mouse input and bit-mapped graphic output. There is an ongoing and valuable need for the prediction and modelling of user activities within such environments. As human-machine dialogues evolve and become more "direct", the processes and limitations underlying our ability to execute rapid, precise movements emerge as performance determinants in interactive systems. Powerful models such as Fitts' law can provide vital insight into strategies for optimizing performance in a diverse design space.

Acknowledgement

This paper is based on research conducted at the Computer Systems Research Institute in the University of Toronto. The financial support of the Natural Sciences and Engineering Research Council of Canada, Xerox Palo Alto Research Center, and Digital Equipment Corp. is gratefully acknowledged.

References

Badler, N. I., Webber, B. L., & Kalita, J. (1991). Animation from instructions. In N. I. Badler, B. A. Barsky, & D. Zeltzer (Eds.), *Making them move: Mechanics, control, and animation of articulated figures* (pp. 51-93). San Mateo, CA: Morgan Kaufmann

Boritz, J., Booth, K. S., & Cowan, W. B. (1991). Fitts's law studies of directional mouse movement. *Proceedings of Graphics Interface '91* (pp. 216-223). Toronto: CIPS.

Card, S. K., English, W. K., & Burr, B. J. (1978). Evaluation of mouse, rate-controlled isometric joystick, step keys, and text keys for text selection on a CRT. *Ergonomics, 21*, 601-613.

Card, S. K., Mackinlay, J. D., & Robertson, G. G. (1990). The design space of input devices. *Proceedings of the CHI '90 Conference on Human Factors in Computing Systems* (pp. 117-124). New York: ACM.

Card, S. K., Moran, T. P., & Newell, A. (1980). *Psychology of human-computer interaction.* Hillsdale, NJ: Erlbaum.

Epps, B. W. (1986). Comparison of six cursor control devices based on Fitts' law models. *Proceedings of the 30th Annual Meeting of the Human Factors Society* (pp. 327-331). Santa Monica, CA: Human Factors Society.

Fitts, P. M. (1954). The information capacity of the human motor system in controlling the amplitude of movement. *Journal of Experimental Psychology, 47,* 381-391.

Fitts, P. M., & Peterson, J. R. (1964). Information capacity of discrete motor responses. *Journal of Experimental Psychology, 67,* 103-112.

Gillan, D. J., Holden, K., Adam, S., Rudisill, M., & Magee, L. (1990). How does Fitts' law fit pointing and dragging? *Proceedings of the CHI '90 Conference on Human Factors in Computing Systems* (pp. 227-234). New York: ACM.

Han, S. H., Jorna, G. C., Miller, R. H., & Tan, K. C. (1990). A comparison of four input devices for the Macintosh interface. *Proceedings of the 34th Annual Meeting of the Human Factors Society* (pp. 267-271). Santa Monica, CA: Human Factors Society.

Hancock, P. A., & Newell, K. M. (1985). The movement speed-accuracy relationship in space-time. In H. Heuer, U. Kleinbeck, & K.-H. Schmidt (Eds.), *Motor behavior: Programming, control, and acquisition* (pp. 153-188). New York: Springer-Verlag.

Hardock, G. (1991). Design issues for line-driven text editing/annotation systems. *Proceedings of Graphics Interface '91* (pp. 77-84). Toronto: CIPS.

Hick, W. E. (1952). On the rate of gain of information. *Quarterly Journal of Experimental Psychology, 4,* 11-26.

Hyman, R. (1953). Stimulus information as a determinant of reaction time. *Journal of Experimental Psychology, 45,* 188-196.

Kurtenbach, G., & Buxton, W. (1991). GEdit: A testbed for editing by contiguous gesture. *SIGCHI Bulletin, 32*(2), 22-26.

MacKenzie, I. S. (1989). A note on the information-theoretic basis for Fitts' law. *Journal of Motor Behavior, 21,* 323-330.

MacKenzie, I. S. (1991). *Fitts' law as a performance model for human-computer interaction.* Doctoral dissertation. University of Toronto.

MacKenzie, I. S. (in press). Fitts' law as a research and design tool in human-computer interaction. *Human-Computer Interaction.*

MacKenzie, I. S., & Buxton, W. (in press). Extending Fitts' law to two dimensional tasks. *Proceedings of the CHI '92 Conference on Human Factors in Computing Systems.* New York: ACM.

MacKenzie, I. S., Sellen, A., & Buxton, W. (1991). A comparison of input devices in elemental pointing and dragging tasks. *Proceedings of the CHI '91 Conference on Human Factors in Computing Systems* (pp. 161-166). New York: ACM.

Meyer, D. E., Smith, J. E. K., Kornblum, S., Abrams, R. A., & Wright, C. E. (1990). Speed-accuracy tradeoffs in aimed movements: Toward a theory of rapid voluntary action. In M. Jeannerod (Ed.), *Attention and performance XIII* (pp. 173-226). Hillsdale, NJ: Erlbaum.

Miller, G. A. (1953). What is information measurement? *American Psychologist, 8,* 3-11.

Newell, A., & Card, S. K. (1985). The prospects for psychological science in human-computer interaction. *Human-Computer Interaction, 1,* 209-242.

Pew, R. W. (1969). The speed-accuracy operating characteristic. *Acta Psychologica, 30,* 16-26.

Rouse, W. B. (1988). Adaptive aiding for human/computer control. *Human Factors, 30,* 431-443.

Shannon, C. E., & Weaver, W. (1949). *The mathematical theory of communication.* Urbana, IL: University of Illinois Press.

Walker, N., & Smelcer, J. B. (1990). A comparison of selection times from walking and pull-down menus. *Proceedings of the CHI '90 Conference on Human Factors in Computing Systems* (pp. 221-225). New York: ACM.

Ware, C., & Mikaelian, H. H. (1989). A evaluation of an eye tracker as a device for computer input. *Proceedings of the CHI+GI '87 Conference on Human Factors in Computing Systems and Graphics Interface* (pp. 183-188). New York: ACM.

Welford, A. T. (1960). The measurement of sensory-motor performance: Survey and reappraisal of twelve years' progress. *Ergonomics, 3,* 189-230.

Welford, A. T. (1968). *Fundamentals of skill.* London: Methuen.

ARCBALL: A User Interface for Specifying Three-Dimensional Orientation Using a Mouse

Ken Shoemake
Computer Graphics Laboratory
University of Pennsylvania
Philadelphia, PA 19104

Abstract

Arcball is an input technique for 3-D computer graphics, using a mouse to adjust the spatial orientation of an object. In Arcball, human factors and mathematical fundamentals come together exceptionally well. Arcball provides consistency between free and constrained rotations using any direction as an axis; consistent visual input and feedback; kinesthetic agreement between mouse motion and object rotation; and consistent interpretation of mouse position. Attention to mathematical detail facilitates the tasks of users and implementors. Users say that as a general-purpose rotation controller Arcball is easier to use than its nearest rival, the Virtual Sphere. It is also more powerful, and simpler to implement.

Résumé

Arcball est une interface utilisateur pour un système de visualisation 3D utilisant la souris pour orienter les objets dans l'espace. Dans Arcball, facteurs humains et concepts mathématiques se marient exceptionnellement bien. Arcball assure la consistence entre rotations libres et contraintes quelle que soit la direction de leur axe. Consistence entre l'entrée et la réponse; accord kinésique entre mouvements de la souris et rotations de l'objet; interprétation consistente des positions de la souris. Les utilisateurs affirment que le contrôleur universel de rotation Arcball est plus simple à utiliser que son concurent direct: «the Virtual Sphere». Il est aussi plus puissant et plus simple à implémenter.

Key Words: object movement, view movement, mouse, user interface, interactive graphics, 3D graphics, rotation, orientation, human factors, quaternion

Introduction

In computer simulated scenes, most objects are manipulated as *if* they were rigid, even when they are rubber balls or robot arms. (Arms are divided into linked rigid pieces.) When one point of a rigid body is fixed by a translation, the remaining variability in position is called the body's orientation. Arcball is an input technique which allows users to adjust orientation using a mouse. The design is unusual in that it considers mathematical fundamentals as well as human factors to address a difficult problem.

Although we find it easy to pick up a small object in one hand, and turn it this way and that to examine it, it has proved much harder to devise a mouse interface which feels nearly as natural. One problem is that changes in orientation can be made in three independent directions—for example, a rotation about a left-right x axis, about an up-down y axis, or about an in-out z axis. In contrast, a mouse can move in only two independent directions.

A deeper problem is the curved geometry of orientation space, which is quite different from the flat space of mouse movements. For example, a 360° rotation leaves the orientation unchanged, while a 180° z rotation gives the same orientation as a 180° x rotation followed by a 180° y rotation. The first case would be like pushing the mouse straight forward and finding it back where it started, while the second would be like raising the mouse in the air by pushing it forward and to the side.

Moreover, closed loops of mouse motion may not produce the closed loops of rotation one expects. Rotate an object by +90° x, +90° y, –90° x, and then –90° y, and it is not back as it started, but off by 120°. This "hysteresis" effect is not inevitable, and would not be tolerated in a translation controller. The point is not that such behavior precludes undoing a complete drag sequence, but that it is unforgiving during dragging. Yet only Arcball avoids hysteresis, by a more careful mapping of mouse input to rotation.

Alternative mouse input techniques were recently evaluated in [Chen 88]. These include simulating sliders or treadmills; selecting a coordinate axis (by menu or by mouse button) then dragging; approximating a trackball; and so on. All separate rotations into independent x, y, and z angles (at least internally), yet psychological studies show that mental models of rotation do not [Carlton 90]. The virtual trackball of [Hultquist 90] computes an instantaneous rotation axis directly, but still exhibits hysteresis.

To avoid hysteresis, we must begin with the mathematical fundamentals. From those, we are immediately led to the basic Arcball design. Then we will see how to augment free rotation with a constraint mode. After pausing to admire an elegant quaternion implementation, we end by evaluating Arcball's success as a general-purpose rotation controller.

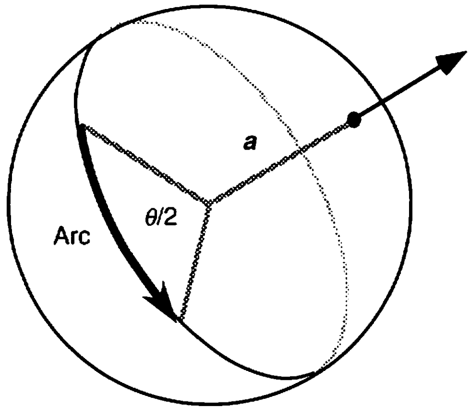


Figure 1. Arc interpretation

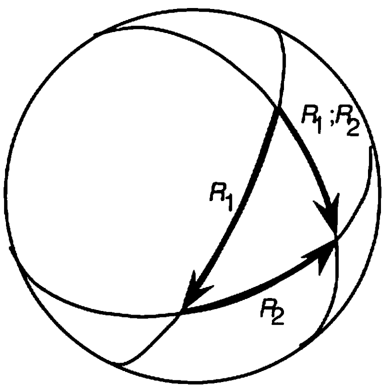


Figure 2. Arc combination

Mathematical Fundamentals

Any orientation of a rigid body can be given by a single rotation, a turn about some axis, starting from an agreed-upon reference orientation. Furthermore, the combination of any number of rotations can be given by a single rotation [Euler 1752] [Goldstein 80].

The combination law takes different forms, depending on the rotation parameters used. Behind the various formulae, however, is simple spherical geometry. A rotation R about axis $\boldsymbol{a}$ by amount θ can be represented on a sphere as any directed arc of length $^1/_2\theta$ in the plane perpendicular to $\boldsymbol{a}$, with positive angles giving a counter-clockwise direction around $\boldsymbol{a}$. (See Fig. 1.) When the end of the first arc is made to coincide with the beginning of the second, two sides of a spherical triangle are formed. The arc completing the triangle, from the beginning of the first arc to the end of the second, represents a single rotation which has the same effect as performing R_1 followed by R_2. (See Fig. 2.) Since rotations do not commute, when the order of combination is reversed, a different arc results. Otherwise, this is like vector addition.

Consider the example given earlier of 180° rotations about x and y giving a 180° rotation around z; this is represented by a 90° arc from the north pole to the equator, followed by a 90° arc along the equator. The third leg of the triangle is indeed a 90° arc, and represents a 180° z rotation. That this is the correct result is easy to verify using a physical object. Furthermore, half-length arcs were essential, as full-length arcs would have predicted a result of no rotation.

The hysteresis example is more complicated, for we must slide arcs around on their great circles in order to add them. First we go from 45° north down to the equator, then 45° east. We then slide the diagonal result arc back by its length, so it ends where we began. We are half done. Now we add a 45° arc to the north pole, giving a result arc also going due north. We slide this down so it ends on the equator, then add a final 45° arc going west. The outcome of all this is a 60° diagonal arc, representing a 120° rotation around the axis (1,-1,-1). This is not at all a simple closed path, but it is how the rotations truly combine.

From these two examples we can see how previous rotation controllers go astray mathematically. Most do not use any kind of spherical model, and those that do rotate "physical" spheres through full-length arcs. Unlike translations, rotations do not admit a choice of "C:D ratio" in this aspect of their model; only half-length arcs are permitted.

The mathematical and physical spheres are similar in that, while arc lengths differ, their planes or axes correspond. While the length problem is surprising, and perhaps confusing, it is also unavoidable. (How many kids have thought adding fractions would be much simpler if we could just add numerators and denominators separately?) For an excellent discussion of the relevant mathematical and physical theory, see Chapter 4 of [Biedenharn 81].

Arcball

Arcball takes its design—and its name—directly from the mathematics. Consequently, rotations can be displayed, as well as input, in a graphically meaningful way.

Suppose some object is selected on the screen. To change its orientation, the user simply draws an arc on the screen projection of a sphere. The arc is a great arc specified by its initial and final points, which are given by the mouse down and up positions. (See Fig. 3.) As the mouse is being dragged, its current position is used to compute the arc. Thus the object—or a faster-drawn stand-in—turns with the mouse. The direction and amount of turn are those of the half-length arc model described above.

A great arc—the shortest spherical path between two points—lies in the plane containing the two points and the center of the sphere. If the end points lie exactly opposite each other (which for Arcball can only happen if they lie on the sphere silhouette), the plane of the arc is not defined. In this application, however, that doesn't matter; opposite points imply a 360° rotation, which leaves the orientation unchanged. In effect, opposite points on the circle are "the same point."

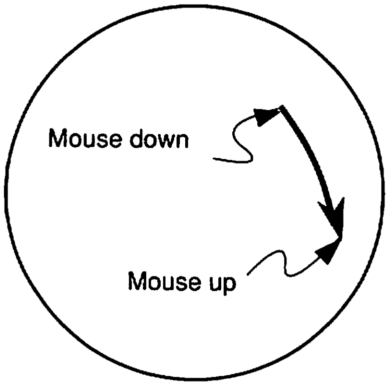


Figure 3. Mouse input

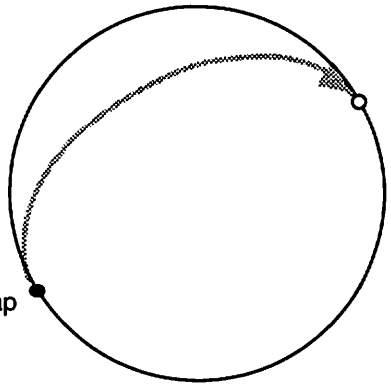


Figure 4. Wrapping.

This permits a useful feature: If the mouse is dragged out of the circle at some point the arc can be made to reenter at—wrap around to—the opposite point. (See Fig. 4.) Although the arc end jumps across the circle, the orientation being controlled changes smoothly and naturally. There is thus no limit on the amount of rotation.

We give the user graphical feedback as the mouse is dragged, by drawing a "rubber-band" arc from the initial point to the current point, and updating a picture of the scene to show the rotation of the body or camera. The sphere and rubber-band arcs can be displayed separately from the scene image, but preferably are drawn transparently on top of it.

Constrained Rotation

Spatial manipulations can be complex, and often are easier with some constraints [Bier 86]. With Arcball, fixed axis constraints are a natural extension. Axes can be chosen from sets of any size, based on the object, the environment, the view, or other sources. Natural axes to allow include the view coordinate axes, the selected object's model space coordinate axes, world space coordinate axes, normals and edges of surfaces, and joint axes of articulated models (such as robot arms).

Remember that every arc lies in a plane perpendicular to its rotation axis. If the two mouse points on the sphere are orthogonally projected onto a fixed plane through the sphere center then radially projected back onto the sphere, they will necessarily become points giving a rotation about the fixed axis perpendicular to that plane. (Fig. 5 illustrates a body z-axis constraint.)

A method must be provided to ask for constraint, and to select the axis to use. Many choice mechanisms, such as menus, are possible, but one is particularly attractive. Superimpose on the sphere a great circle (the front half) for each axis from a limited set. Thus for body coordinate axes, three mutually perpendicular arcs would be drawn, tilted with the object. When the mouse is clicked down to initiate a rotation, the constraint axis selected—and the only one shown—will be that of the nearest arc. So that the user doesn't have to guess, we dynamically highlight the nearest arc as the mouse is moved around prior to clicking. Mouse, menu, or keyboard combinations can be used to select among axis sets (e.g., SHIFT-click for camera axes, CTRL-click for body axes, unmodified click for no constraints).

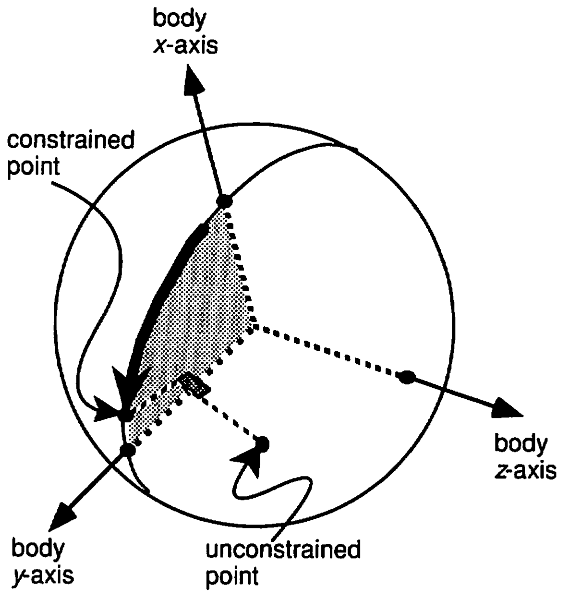


Figure 5. Constrained z rotation

Implementation

All the code for Arcball is here. The sphere can be indicated by drawing its silhouette circle. To transform cursor coordinates on the screen into a point on the sphere, the center of the circle is subtracted from the cursor coordinates giving a radial vector, which is divided by the radius of the circle to give two of the coordinates on the unit sphere. If the cursor lies outside the circle, that is easily corrected now. The third sphere coordinate is obtained as the quantity which makes the sum of the squares 1. Specifically, let the cursor screen coordinates be `screen.x` and `screen.y`, let the center of the circle be at `center.x` and `center.y`, and let the radius on the screen be `radius`. Then the coordinates on the sphere are given by

```
pt.x ← (screen.x - center.x)/radius;
pt.y ← (screen.y - center.y)/radius;
r ← pt.x*pt.x + pt.y*pt.y;
IF r > 1.0
    THEN s ← 1.0/Sqrt[r];
         pt.x ← s*pt.x;
         pt.y ← s*pt.y;
         pt.z ← 0.0;
    ELSE pt.z ← Sqrt[1.0 - r];
```

When a constraint axis is being used, the sphere point is projected onto the perpendicular plane, flipped to the front hemisphere if necessary, and renormalized before being used. If the point lies on the axis, an arbitrary point on the plane must be chosen. Flipping exploits the wrap effect.

```
dot ← V3_Dot[pt, axis];
proj ← V3_Sub[pt, V3_Scale[axis, dot]];
norm ← V3_Mag[proj];
IF norm > 0
    THEN s ← 1.0/norm;
         IF proj.z < 0 THEN s ← -s;
         pt ← V3_Scale[proj, s];
ELSE IF axis.z = 1.0
    THEN pt ← [1.0, 0.0, 0.0];
ELSE pt ← V3_Unit[[-axis.y, axis.x, 0]];
```

Incidentally, to find the closest arc simply constrain with each axis in turn, and pick the one that gives the nearest point. The nearest constrained point will have the largest dot product with the free point.

Having obtained the initial and final end points by this means, the rotation in unit quaternion form [Shoemake 85] is the product of the final point times the conjugate of the initial point: $q = p_1 p_0^*$. Essential quaternion facts are: (1) a unit quaternion $q = [v, w] = [x, y, z, w]$ consists of a scalar w which is cos $\theta/2$ (where θ is the rotation angle), and a vector v which is sin $\theta/2$ times a unit vector along the rotation axis; and (2) the product of two quaternions gives the combination of the rotations they represent, and is non-commutative. The formula given amounts to setting the quaternion vector to the cross product of the initial and final points, and the quaternion scalar to their dot product. No trigonometric functions are required, only simple arithmetic.

```
[q.x, q.y, q.z] ← V3_Cross[p0, p1];
q.w ← V3_Dot[p0, p1];
```

The new orientation of the rigid body is given by the product of the quaternion for the orientation when dragging started with the quaternion we've just derived from the user's mouse input:

```
qnow ← QuatMul[q, qstart];
```

This product uses only 16 multiplies and 12 adds, and the result can be converted to a matrix at about the same cost [Shoemake 89]. If a graphics pipeline with 4×4 matrix multiply is available, accumulation and conversion can be done at essentially no cost [Shoemake 92].

From these quaternion formulae we can verify that spherical triangles behave as described earlier. Arc lengths will be half the rotation angle, since we rotate by twice the inverse cosine of the dot product, which is cosine of the arc length. Also, the axis of rotation is taken from the cross product, which is perpendicular to the vectors from the center to the end points. To go from p_0 to p_1, we use $p_1 p_0^*$, and we continue from p_1 to p_2 using $p_2 p_1^*$. The combined effect is given by the product $p_2 p_1^* p_1 p_0^*$, which for these unit quaternions is equivalent to $p_2 p_1^{-1} p_1 p_0^{-1} = p_2 p_0^*$, corresponding to the third leg of the triangle. We can also see that opposite points are equivalent, since $-q$ gives the same rotation as q.

For many purposes, the unit quaternion is most convenient, however a quaternion can easily be converted to other forms [Shoemake 85]. Although modern systems use quaternions already, an Arcball implementation can certainly be done without them. The essential step will still be to compute the dot product and cross product of the arc endpoints.

A few observations may help simplify the arc drawing code. We can approximate an arc with N line segments if we know an endpoint and a point $1/N$ of the arc length away, by reflection. Reflect the first point across the second, then the second across the third, and so on. If δ is the dot product of p_t and $p_{t+\Delta}$, then $p_{t+2\Delta} = 2\delta p_{t+\Delta} - p_t$. We find δ just once, then ignore the z coordinates. Given the endpoints, p_0 and p_1, and the arc length, $\Omega = \cos^{-1} p_0 \cdot p_1$, we can compute the extra point as $(p_0 \sin((N-1)\,\Omega/N) + p_1 \sin \Omega/N)/\sin \Omega$. An arc for constraint axis $\boldsymbol{a} = [x, y, z]$ is split in two. If $s = \sqrt{1-z^2}$ is non-zero, the endpoints will be $[-xz/s, -yz/s, s]$ and $\pm[y/s, -x/s, 0]$; otherwise the "arc" is the sphere silhouette.

Orientation graphing

To go the other way, from a unit quaternion to a pair of points on the sphere, first pick an initial point—say a point on the sphere edge—perpendicular to the quaternion vector; then obtain a final point as the product of the quaternion times the initial point. The following suffices.

```
s ← SqRt[q.x*q.x + q.y*q.y];
IF s = 0.0
    THEN p0 ← [0.0, 1.0, 0.0]
    ELSE p0 ← [-q.y/s, q.x/s, 0.0];
p1.x ← q.w*p0.x - q.z*p0.y;
p1.y ← q.w*p0.y + q.z*p0.x;
p1.z ← q.x*p0.y - q.y*p0.x;
```

Also, if desired, the initial rim point can be negated when that would give a shorter arc:

```
IF q.w < 0
    THEN p0 ← [-p0.x, -p0.y, 0.0];
```

Evaluation

At this point in the paper, it's a good bet that the graphics programmers and mathematicians are happy, but the readers who go to SIGCHI conferences are wondering what became of the user in this user interface. Their concern is legitimate, for as [Gentner 90] observed, "good engineering practice can lead to poor user interfaces."

A good user interface is quickly learned and easily remembered; gets the task done quickly and with few errors; and is attractive to users [Foley 90]. These goals are more likely to be met if the design uses simple and natural interactions in "the user's language," requires little memorization, provides feedback and shortcuts, and is consistent—both with itself and other interfaces [Nielsen 90].

How well does Arcball meet these criteria? We can make a preliminary assessment before looking at user tests. It is manifestly simple to use. There are no multiple sliders, knobs, or mouse buttons. Selecting constraints is as simple as holding down a key and clicking on an arc.

Dragging on a sphere is fairly natural. The object turns the same direction the mouse moves, for a natural kinesthetic correspondence. We can repeat an orientation by repeating a position, as holding physical objects leads us to expect. On the other hand, half-length arcs are probably not in "the user's language," so are a potential problem.

Users must remember to hold down a key to get constraints, but they never have to remember which direction is object, world, or view x, y or z. The interface itself quickly reminds users of how it behaves. (As for remembering constraints, it may be helpful to add a persistent constraint mode, toggled with a menu item.)

Arcball is rich in feedback—more so than most controllers. The object rotates for a sense of direct manipulation, and a rubber-band arc shows the net effect of a drag. Constraint mode is signaled both by muscle tension [Buxton 86], and a visual display of arc choices. The arc to be selected is highlighted, for further constraint feedback. The orientation of the object can be graphed as yet another feedback arc. It is, of course, possible to augment Arcball with numeric output (and input) when that is meaningful.

Arcball hardly needs shortcuts, but there are three. First, the availability of constaints can be considered a shortcut for times when they are desirable. (Constraints can also be used as "training wheels.") Second, wrap-around makes large rotations easier; but notice that half-length arcs already let the user rotate by 360° around any axis with a single drag. Third, dragging outside the circle is an easy way imitate the use of a screen normal constraint axis. Angle detents can be added to make, say, 15° angle multiples easier.

Consistency is one of Arcball's strongest features. Object motion is consistent with mouse motion, as noted, and lack of hysteresis is a very powerful kind of consistency. Also, at any time within any drag, mouse movement between the same points will always turn the object exactly the same way. Arcs will be drawn for dragging, for constraints, and for object orientation; all are interpreted within a consistent context. Some interfaces have "trouble spots," such as gimbal lock, at certain angles; Arcball has none. Interfaces like the Virtual Sphere that depend on incremental mouse movements can behave badly if sampling is slow or coordinates are noisy; Arcball will not. There is little difference between use of the mouse in constraint mode and free mode, so Arcball can be a general-purpose controller.

The Arcball consistency of constrained rotation with free rotation means users need learn only one interface for many purposes. Consider the manipulation of a manikin arm with rotary joints [Badler 86]. The shoulder has a ball joint and so can rotate freely, while the elbow can bend around only one axis. With other interfaces, a different style of input might be required for each joint. A thumbwheel is a common choice for the elbow, but using three wheels for the shoulder is awkward. Even for the elbow, the relationship of wheel motion to spatial motion may change when the shoulder is rotated, so a horizontal stroke of the wheel corresponds to a diagonal bend of the elbow. Arcball eliminates all these difficulties, and simplifies new possibilites. For example, the foot can be forced to pivot on a sloping floor without raising the heel or toe, simply by constraining its rotation to be around the normal to the floor.

User interfaces cannot be evaluated adequately on paper, yet empirical measures of advantage are often hard to obtain. Even the study in [Chen 88] failed to find significant performance differences between two very different controllers, and only concluded the superiority of the Virtual Sphere over the method of [Evans 91] based on user comments. Yet informal evaluation should not be quickly dismissed, as studies have shown it can be informative [Nielsen 90].

Informal tests of Arcball suggest that its visual feedback provides important cues for understanding its behavior, that it is valuable to have both free and constrained modes, and that eliminating hysteresis is helpful. Arcball rotates objects twice as far as might be expected, yet few users realized that; when they did, it was not a problem. A number of users were slow learning how to rotate around the screen normal. This confusion was possible because they were given no initial hint about how Arcball worked, in order to learn as much as possible about their expectations and responses. Nevertheless, without exception, they quickly learned to use, and like, Arcball. Many described it as "a sort of trackball." Trackballs usually have only two degrees of freedom, which may explain the difficulty mentioned.

Further comment on visual feedback is in order. Unlike other controllers, Arcball can draw a meaningful rubber band arc while dragging, but it was not clear whether it was worth cluttering the picture. In fact, users appreciated the arc, and noted that it was a strong cue to the spherical nature of the controller. The circular silhouette of the controller sphere might also be eliminated; however without it, Arcball is much more difficult to use. Finally, a constraint axis can be chosen as the nearest arc from some set; not surprisingly, it is better to draw the arc choices, and to highlight the closest one before the mouse is clicked to begin dragging.

When specifically compared to Chen's Macintosh demo of the Virtual Sphere, Arcball was the clear favorite.[†] Hysteresis may be hard to describe in writing, but users easily noticed the different feel when they tried the controllers together, and preferred the feel of Arcball. The Virtual Sphere had a bothersome modal distinction between drags that started inside the circle and those that started outside; Arcball did not. Use of the shift key (consistent with some other Macintosh interfaces) invoked a limited constraint mode for the Virtual Sphere, but there was no visual feedback, and users found its behavior hard to understand. Some users were slow to notice the Arcball arc highlighting (a more distinct color should probably be used), but since only the selected arc remained visible when dragging began, they could see their mistakes. In this area, too, Arcball was preferred. As mentioned earlier, half-length arcs were unexpected, but not annoying. Their benefits of wrap-around, greater range of motion, arc feedback, and hysteresis elimination seemed more important than their lack of "physicality."

Conclusions

Arcball is an elegant application of mathematical theory to interface design. Its behavior and its implementation are clean and simple. We can perform both free rotation and constrained rotation. In either case, the direction of mouse motion corresponds to the direction of object rotation. Lack of hysteresis makes Arcball more forgiving than other rotation controllers, since incremental motions are easily undone. Use of half-length arcs brings this and other benefits, without seeming unnatural. More user studies are needed, but mathematically, at least, Arcball is likely to be the best general-purpose rotation controller using a mouse.

That said, there is a bigger picture. Arcball only controls rotation, and typical 3-D interactions certainly require translation [Houde 92] and possibly scaling as well—and along arbitrary axes [Shoemake 92]. Studies indicate important differences between manipulating objects and manipulating views [Ware 90]. For the latter, egocentric controllers [Mackinlay 90] may be more satisfactory. Users may feel that consistency of controllers should encompass all three transformations, or may decide that specific tasks warrant custom rotation controllers. Ultimately, users will choose the interfaces that serve them best.

Since a single mouse position has only two degrees of freedom, a pair of positions—the ends of an arc—are used. This part of Arcball has wider applicability, including a translation controller to be described in a future paper.

[†] Readers who would like to make the comparison for themselves can get a copy of the Arcball demo for the Macintosh by anonymous ftp to ftp.cis.upenn.edu, directory /pub/graphics.

Acknowledgments

Xerox PARC proved a fertile ground for the blooming of these ideas. Jules Bloomenthal helped me integrate Arcball into existing software. Jock Mackinlay, George Robertson, and Eric Bier gave me enthusiastic support, often at some sacrifice, for which I am very grateful. Thanks to all the hardy folk who took the time to try Arcball and give me their comments. Pierre Crégut translated the abstract.

References

[Badler 86] Badler, Norman I., Manoochehri, Kamran H., and Baraff, David. "Multi-Dimensional Input Techniques and Articulated Figure Positioning by Multiple Constraints," in Pizer, Stephen M., ed., Proceedings 1986 Workshop on Interactive 3D Graphics, ACM, 1986, 151–169.

[Biedenharn 81] Biedenharn, L.C., and Louck, J.D. *Angular Momentum in Quantum Physics: Theory and Application.* As *Encyclopedia of Mathematics and Its Applications*, Gian-Carlo Rota (ed.), Vol. 8. Addison-Wesley, 1981.

[Bier 86] Bier, Eric A. "Skitters and Jacks: Interactive 3-D Positioning Tools," *Proceedings 1986 Workshop on Interactive 3-D Graphics* (Chapel Hill, North Carolina, October 1986), 183–196.

[Buxton 86] Buxton, William. "There's More to Interaction Than Meets the Eye: Issues in Manual Input," in Norman, D., and Draper, S., eds. *User-Centered System Design*, Lawrence Erlbaum, 1986, 319–337.

[Chen 88] Chen, Michael, Mountford, S. Joy, and Sellen, Abigail. "A Study in Interactive 3-D Rotation Using 2-D Control Devices," *Computer Graphics*, 22 (4), August 1988 (SIGGRAPH '88 Proceedings), 121–129.

[Euler 1752] Euler, Leonhard. "Decouverte d'un nouveau principe de méchanique," (1752), *Opera omnia, Ser. secunda, v. 5*, Orel Füsli Turici, Lausannae, 1957, 81–108.

[Evans 81] Evans, Kenneth B., Tanner, Peter P., and Wein, Marceli. "Tablet Based Valuators that Provide One, Two, or Three Degrees of Freedom," *Computer Graphics*, 15 (3), August 1981. (SIGGRAPH '81 Proceedings), 91–97.

[Goldstein 80] Goldstein, Herbert. *Classical Mechanics, second edition*, Addison-Wesley, 1980.

[Houde 92] Houde, Staphanie. "Iterative Design of an Interface for Easy 3-D Direct Manipulation," *CHI '92 Conference Proceedings* (Montery, California, May 3–8, 1992), ACM 1992.

[Hultquist 90] Hultquist, Jeff. "A Virtual Trackball," *Graphics Gems*, Academic Press, 1990, 462–463.

[Mackinlay 90] Mackinlay, Jock D., Card, Stuart K., and Robertson, George G. "Rapid Controlled Movement Through a Virtual 3D Workspace," *Computer Graphics*, 24 (4), August 1990. (SIGGRAPH '90 Proceedings), 171–176.

[Nielsen 90] Nielsen, Jakob, and Molich, Rolf. "Heuristic Evaluation of User Interfaces," *CHI '90 Conference Proceedings* (Seattle, Washington, April 1–5, 1990), ACM 1992.

[Shoemake 85] Shoemake, Ken. "Animating Rotation with Quaternion Curves," *Computer Graphics*, 19 (3), July 1985. (SIGGRAPH '85 Proceedings), 245–254.

[Shoemake 89] Shoemake, Ken. "Quaternion Calculus For Animation," Notes for Course #23, *Math for SIGGRAPH*, SIGGRAPH '89.

[Shoemake 91] Shoemake, Ken. "Quaternions and 4¥4 Matrices," Graphics Gems II, Academic Press, 1991, 351–354.

[Shoemake 92] Shoemake, Ken. "Matrix Animation and Polar Decomposition," *Graphics Interface '92 Proceedings* (Vancouver, British Columbia, May 11–15, 1992).

[Ware 90] Ware, Colin and Osborne, Steve. "Exploration and Virtual Camera Control in Virtual Three Dimensional Environments," *Computer Graphics* 24 (2), March 1990, 175–183.

Designing Video Annotation and Analysis Systems

Beverly L. Harrison
Department of Industrial Engineering and
Dynamic Graphics Project, Computer Systems Research Institute
University of Toronto

Ronald M. Baecker
Department of Computer Science and
Dynamic Graphics Project, Computer Systems Research Institute
University of Toronto

Abstract

Although video has been used for many years to record data, few tools have been developed to help analyze video data. Current multimedia interfaces have severe cognitive and attentional limitations, reflecting technology-centred designs which have not profited from human factors theory and user-centred design. This paper discusses user requirements for video analysis, from which we derive a set of functional specifications. These specifications are useful for evaluating existing systems and for guiding the development of new systems. A number of existing systems are briefly described, as is the VANNA Video ANNotation and Analysis system, which integrates video, non-speech audio, voice, textual and graphical data, and which incorporates emerging technology, user-centred design and human factors theory.

Keywords

Video annotation, video analysis, analysis methodologies, usability testing, multimedia.

Introduction

Video is a vivid and compelling way of presenting information, of highlighting interesting experimental findings, and of illustrating unique concepts. It provides a highly-detailed, permanent record which can be analyzed in many ways to extract a variety of different types of information. These benefits have long been recognized in usability testing, training and education (e.g., Anacona, 1974; Dranov, Moore, and Hickey, 1980; Ramey, 1989; Nielsen, 1990). New applications in video mail (e.g., Buxton and Moran, 1991), interactive multimedia systems (e.g., Ishii, 1990; Mantei, Baecker, Sellen, Buxton, Milligan, and Wellman, 1991), behavioral research and computer supported cooperative work (e.g., Greif, 1988; Baecker, 1992) are driving an increasing demand for better tools for handling and analyzing video.

Consider the following example. In one typical video analysis application experimenters video taped two subjects in a collaborative programming task. The subjects communicated using a video link between their two distant locations. Experimenters were interested in noting when subjects were looking at the computer monitor and when they were looking at the video monitor. They additionally wanted information about whether subjects looked or did not look at the video image of their partner at times when they communicated. Finally, experimenters wanted to know what types of information were communicated and passed between the two subjects using the video link (e.g., diagrams, pointing to parts of the computer screen or manual, normal conversational gestures). The data resulting from such an analysis might include gaze information as it related to conversation, and data about information content.

For video analysis tools to succeed they must support a wide variety of tasks and analysis styles while still easily capturing essential data. We need to gain new insights into the way people work with multimedia systems. Little is known about creating new classes of interface which manipulate information having temporal dependencies. We need to apply proven design methodologies, human-computer interaction and human factors theory.

The objective of this paper is to provide the reader with an understanding of design issues for video analysis systems, using the VANNA system as an illustrative implementation of one such system. The first section of this paper describes user requirements for video analysis, enabling us to derive a set of functional specifications for building video analysis tools. These specifications have been applied to evaluate existing "landmark" tools and notation systems (e.g., Rein, 1990; Losada and Markovitch, 1990; Potel and Sayre, 1976; Roschelle, Pea, and Trigg, 1990) and suggest guidelines for the development of new multimedia tools. We then describe the VANNA system, which reflects these guidelines and illustrates a number of unique interface design approaches. Tests results for the VANNA system are presented and design implications are discussed.

User Requirements

Our intent is to provide a multimedia video analysis tool which is easily customized to address the characteristics of the task, the application and the user's personalized style. To achieve this task analyses were performed for multiple users within single application domains and across many different application domains (e.g., usability testing, CSCW, behavioral studies). We also conducted literature reviews and surveys, examined existing systems used for video analysis, and interviewed users of these systems to determine which functionality the systems had in common, which functions were most frequently and least often used, and what the common complaints were. (See Harrison and Baecker, 1991; Harrison, 1991 for detailed discussions of the systems examined.) A summary of the most important results of this work is presented here.

From the task analysis, we derived two key points related to the *process* of manipulating a video document. Users tend to work with video in one of two ways: *annotation* and detailed *analysis*. Annotation implies "note taking." Here users are attempting to capture data in real-time, in highly personalized and abbreviated ways. The annotation task is characterized by high cognitive and attentional demands. Detailed analysis typically occurs after the real-time annotation and does not have the same real-time constraints. In this case the user may make many passes over a given segment of tape in order to capture verbal transcriptions (protocol analyses), behavioral interactions, gestural or non-verbal information. As part of this detailed analysis, users may also wish to run statistical analysis, or summarize data in tables or graphs.

Based on the user interviews and surveys of existing systems, we derived a set of user requirements which support *both* the annotation and the detailed analysis process. These were grouped into four categories: coding the data, analyzing and interpreting the data, user interface and device control, and displaying the data. The *coding* category represents methods for entering the various forms of annotational and analysis data. Elements in the *analysis and interpretation* category are those which related to manipulating pre-recorded data, in order to form conclusions about the nature of the data. The *user interface and device control* category embodies some general principles for building user interfaces of video annotation and analysis systems. Finally, when *displaying the data*, there are several general requirements to guide presentation formats and capabilities.

Coding the Data

There are two kinds of coding: real-time or on-line coding which occurs during annotation, and off-line coding, which occurs during analysis. On-line or real-time coding may be thought of as a subset of the overall coding process, where the video may be viewed playing forward at normal speed only, with no opportunity for review. A restricted set of functions is used, which reflects the real-time constraints and high attentional demands. These capabilities allow the user to mark events (typically with a single button press, mouse click, or keyboard stroke), and allow entry of very short text comments. The user must be able to:

- mark the occurrence of an event
- mark the start and stop points for intervals.

The system must be able to:

- capture keystrokes
- capture subject's computer screen.

The "off-line" coding process requires a more comprehensive set of functions. This stage is characterized by the high usage of speed control and reviewing capabilities, and permits the user to perform detailed coding operations of data including:

- user comments or general observations
- verbal transcriptions of the conversation
- non-verbal information, e.g., gestures
- personality or mood measures.

Our current findings indicate that user comments are typically fairly concise (estimated at less than 200 characters for text, or 2-3 figures for graphics). Verbal transcriptions of the conversation may be word-for-word transcriptions or might simply record specific spoken keywords. Non-verbal or gestural information may be described in a number of ways, including sketches, special coding schemes or symbols, and may even be embedded in the conversational record. This information is the most difficult to represent and is therefore most often subject to encoding schemes, as demonstrated by some of the current notations (e.g., Heath, 1986, 1990). Personality measures or mood assessments are often also ranked and encoded. Most "mood" coding schemes in current tools are based on the Bales SYMLOG system for studying small group dynamics (Bales and Cohen, 1979). Several mood notation systems currently exist though none integrate video directly into the analysis tool (e.g., Losada and Markovitch, 1990; Rein, 1990).

Analyzing and Interpreting the Data

Once the video has been coded, any number of analyses may be applied to the data. The level of analysis is dependent upon the experiment objectives, hypotheses and experimental design, but some general capabilities are summarized below:

- play "next" event
- play "previous" event
- group events
- play entire group
- play loopback i.e., play the same sequence over many times
- keyword searching for text data
- basic quantitative data – frequencies, averages, durations, variances
- time series or interaction analyses
- data exporting
- merging of data for interjudge reliability.

Interaction patterns play a significant role in many analyses. These patterns can be derived by statistical means, by time series analysis or by approximation through visual inspection of carefully formatted output. This last case provides a more simplistic view of the interactions by summarizing data on adjacent, aligned time lines. This allows users to visually inspect the data for recurring patterns, overlaps and gaps in interaction. Time lines facilitate the observation of *process* information (as opposed to *content* information).

User Interface and Device Control

The user requirements and the attentional demands of video analysis have direct implications for both the user interface and the mechanisms for device control. Critical interface issues include integration of the video images, device controls, and tool functionality, use of both auditory and visual feedback cues, consistency in the interface, and user-definable screen layouts. Technology issues include the degree of a user's control over the video devices and the choice of input devices.

Integration of the "video monitor" with the annotation system on a single screen is crucial because video requires continuous visual attention; important events might be missed in a split second. This continuous monitoring is required since one can neither predict the frequency of event occurrences, nor the modality for the event (i.e., in which channel the event will occur: auditory or visual). Additionally, spatially separated displays (as are prevalent with existing systems, e.g., Losada and Markovitch, 1990; Roschelle, Pea, and Trigg, 1990) prevent simultaneous access to multiple visual sources. Users must direct their visual attention away from the critical data source in order to locate and select functions in the analysis tool. The resulting visual scan time (and effort) between displays is unacceptable for analysis tasks, in particular for real-time annotation. Finally, integration of displays solves problems with work space size limitations vs. equipment size requirements.

The annotation system should have both auditory and visual feedback mechanisms. If the user is analyzing visual data the auditory feedback cues from the system would be used and vice-versa. This minimizes interference between system feedback and the primary task of analyzing the video data. Visual channels are typically differentiated in terms of spatial separation (i.e., different locations in the visual field). Auditory cues are differentiated by pitch, loudness, and tonal characteristics. The auditory cues should be non-speech to avoid confusion with the voice track of the video document.

In order to successfully record detailed events, comments and information, the user requires automatic control of many of the video speeds from within the analysis tool. The minimum speed control requirements are:

- high speed, e.g., fast forward
- regular playing speed
- frame by frame
- paused at any single frame.

Forward and reverse motion options should be applicable to any of these speeds.

The tool must be capable of coding at a variety of temporal "resolutions". This allows events to be coded at a variety of rates, such as:

- every frame
- every second
- every minute
- at random intervals.

If multiple tapes are used for recording, users may need to automatically cue up any or all of the tapes relative to the position of a single tape.

The coding process, and in particular the real-time annotation, has implications for the style of interface and the input devices chosen. Users need to access the various capabilities of the tool with interfaces which have low visual attentional demands. The kinds of mechanisms might include button presses, touch typing, the ability to point directly to the monitor using a touch screen or draw directly using a stylus. It may be desirable for interface mechanisms and graphic annotations to be overlaid directly on top of the video. The interface should avoid secondary monitors and graphics which require fine motor coordination for function selection. Additionally the mechanisms for representing the data should not require complex encoding schemes or cognitive mappings, but rather should favour a direct one-to-one mapping between the concept to be representing and the interface object (and corresponding label) used to capture the item.

Critical in the usability of any tool is the ability to customize the screen. Users must be able to add or delete instances of objects to reflect their current analysis needs. This includes the ability to modify the tool in an ad-hoc manner during an analysis session. Providing a library of functions from which the users can "copy" and "paste" interface objects and subsequently resize and relocate them on the screen is one method of achieving this.

When reviewing previously recorded data, users must be able easily to play back the previous item, the next item, and user-defined groups of items, independent of their actual location in the data file.

Displaying the Data

The presentation of data and results is perhaps one of the most under-developed aspects of current tools. A minimum requirement is the ability to print a copy of all data recorded. This is basically a "dump" of a log file, containing reference time codes or frame locations, comments, transcriptions, diagrams, event markers, interval markers and keystroke logs. (Many tools do not extend their "output" capabilities much beyond this.) The result is a complex listing of data which is usually so dense that interpretation is difficult. The implication of this is the need for a variety of views or summaries of the results. This includes numerical analyses, time line representations and graphical plots. The user should be able to specify whether the analysis results are presented by experimental subject, by topic discussed, by artifact usage or by other criteria.

Most existing tools present results and data in either tabular format or on a text-based time line (either horizontal or vertical). The use of colour and animation contributes greatly to the clarity and effectiveness of presentation. These techniques have been greatly underutilized thus far.

Often users wish to present short video segments which highlight interesting findings or which provide good representations of general trends in the data. In order to achieve this, they need to be able to indicate the starting and stopping points for a number of sequences, and the order in which the sequences are to be played back. These sequences may be existing intervals marked in the coding stage, or they may be new sequences. The order of presentation of sequences should not be dependent upon the video recording medium, i.e., the tape media must support non-sequential playback.

Functional Specifications

From the user requirements a number of functional specifications may be inferred. These have specific implications for tool functionality and for the user interfaces to video analysis systems.

Coding the Data

1. *User-specified indexing of the video tape.*
Users may mark an event or an interval by indicating the starting (and stopping) position, typically by a button press. Still frames may also be used as index markers. These events or intervals can later be retrieved under computer control.
2. *Grouping of events or activities.*
Users can group similar events or activities together in user-specified classes and assign a unique index to each class.
3. *Experimenter observations or comments.*
Users can enter textual or possibly graphical comments, notes and observations. These are linked automatically to the appropriate segments of video.
4. *Verbal transcript analysis and keyword indexing.*
Users can enter conversational transcriptions of the audio track. Statistics may be computed on the keywords.
5. *Individual and group characteristics.*
Users can enter subjective assessment data for various measures of personality and group dynamics such as Bales measures (Bales, 1950).
6. *Non-verbal and gestural information.*
Users can enter data related to the observed gestural patterns, for corresponding video frames or segments. This information may be coded using a variety of notations, including symbolic notations such as Labanotation (Laban, 1956; Hutchinson, 1954).

Analyzing and Interpreting the Data

7. *Keyword searching.*
Users can use keyword searches on any text data, including experimenter comments or verbal transcriptions.
8. *Keystroke and computer screen integration.*
If the subjects are required to use a computer, their keystrokes and/or computer screen is recorded and synchronized with the video.
9. *Access to text editors, statistics packages, graphics packages, plotting packages.*
Users can import/export data to/from other software packages.
10. *Analysis for interaction patterns over time.*
Users can analyze the data by examining patterns in the occurrence of events or activities over time. Significant (frequent) patterns and interactions are highlighted on a time line.
11. *Support for interjudge reliability.*
A means of merging multiples codings should be available to support multiple judges and hence improve reliability of the data and subsequent analysis.

User Interface and Device Control

12. *Digital control access to basic video functions.*
Users can stop, start, fast forward, and rewind video tape(s) and control the playback speed directly from the analysis tool.
13. *Retrieval and playback of previous and next indexed items.*
Based on the currently selected item or the current location on the tape, users can elect to play the next or previous item recorded.
14. *Retrieval and playback of sets of items using automatic indexing.*
Users can request that *all* events or activities belonging to a given class be played in sequence automatically.
15. *Direct manipulation interface.*
Users access the various capabilities of the tool using interfaces which have low visual attentional demands. The kinds of mechanisms might include button presses, touch typing, the ability to point directly to the monitor using a touch screen or draw directly using a stylus. It may be desirable for interface mechanisms and graphic annotations to be overlaid on top of the video. The interface should avoid secondary monitors and graphics which require fine motor coordination for function selection.
16. *Simplified mental models.*
The users should have minimal mappings and coding schemes to represent events, activities and attributes.
17. *Ability to customize annotation screens.*
Users have access to a "library" of functions, from which a subset may be chosen and laid out on a screen to form user-definable interfaces. Users can relocate and resize any object on the screen.
18. *Multi-media or hypermedia analysis record.*
The final analysis record consists of text, audio, and video is integrated into a single multi-media document.
19. *Automatic synchronizing mechanism for multiple tapes.*
If multiple tapes are used for recording, users can automatically cue up any or all of the tapes based on the position of a single tape.

Displaying the Data

20. *Customizable presentation and summarizing capabilities.*
The number and levels of mappings and coding schemes are minimized to facilitate interpretation of data and results. Results should be presented in user-defined categories. Users specify which items to include or exclude in each summary view. Several standard views are provided. Textual and graphical formats are both available.
21. *Time line display of events.*

Users can view the occurrence of events and duration of activities on time lines. Users can specify the number of time lines and the basis on which they are defined (e.g., per subject, per task, per medium)

22. *Animated and colour displays.*
Animation and movement patterns can be used to illustrate dynamics and capture the temporal dimension of behaviour. Colour can be used to distinguish and highlight variables or interesting results.

23. *Presentation of video segments.*
Users can mark video segments which illustrate relevant or interesting examples and produce an "edit list". This edit list can be easily played back in any sequence.

State of the Art

At the time of this research, there were several interesting video analysis systems in existence, each providing a unique contribution to the field. The tools described may be divided into two categories: notation systems and video analysis tools. Notation systems are methods of representing information extracted from video tape, though they may not be directly linked to or control the video itself. Video analysis tools control the video and integrate functionality such as described in the previous section.

Notation Systems

The Heath Notation System is one of the few encoding schemes which directly integrates detailed information about non-verbal communication and gestures with the corresponding verbal transcripts (Heath, 1986; Heath, 1990). Typed punctuation symbols (e.g., ----, ... , ,,,,,, [,]), represent non-verbal events and activities. Information about intonation, speaking volume and speech characteristics is embedded directly into the transcript, while gestures, gaze and other non-speech information is represented above each line of verbal transcript. This analysis method facilitates observation of the interrelations between non-verbal communication patterns and verbal conversation, although the encoding scheme is complicated and makes accurate keyword searches difficult.

The Mood Meter system is a graphical notation system which is based on the Bales SYMLOG dimensions (Bales and Cohen, 1979) and which describes human interaction and mood over time (Rein, 1990; Olson and Storrosten, 1990). The participants' "mood" ratings are aggregated into a single group score, which is represented diagrammatically by concentric circles or stars of varying colour and density. The idea is to represent divergent groups by dispersed images and convergent groups by concentrated images. One drawback of this tool is the reliance on cognitive mapping schemes for encoding participation and group mood. The mood data require interpretation to convert them to the Bales dimensions, followed by translation to descriptive terms. Additionally, the mood diagrams reflect the *aggregate* group mood, making interpretation on an individual level difficult.

Analysis Tools

GALATEA is "an interactive animated graphics system for analyzing dynamic phenomena [notably biological cell movement] recorded on a movie film" (Potel and Sayre, 1976; Potel, Sayre and MacKay, 1980). In this system computer graphics or animated images are superimposed *directly* on the film. Users have the ability to "write directly" on the film with a digitizing stylus over the video screen image, giving Galatea a unique and truly direct manipulation interface. This input mechanism allows free hand drawing, data point entry, or handwritten notes. There is also a substantial easy-to-use button interface to the video controls which resides on the same monitor as the video image, though is not visible simultaneously.

The GroupAnalyzer is one of the most sophisticated tools for representing group dynamics (mood) over time (Losada and Markovitch, 1990; Losada, Sanchez, and Noble, 1990). The presentation capabilities of this tool are exceptionally good, taking advantage of colour, animation and time series analysis. The analysis component allows users to display both static and dynamic (animated) displays of the results in "field diagrams." Users may display an animation demonstrating how the group dynamics evolve and change over time (the dominance circles for each participant expand and contract). The field diagram may be used to reference the actual video tape. Entering the data requires training, however, since the coding forms are complex and make extensive use of cognitive mappings (per the Bales dimensions). Much of the coding is done in real time by trained experimenters while they are observing subjects.

VideoNoter is a tool which allows users to create annotations and verbal transcriptions which automatically index either a video tape or disc (Roschelle, Pea, and Trigg, 1990; Trigg, 1989). Annotations may be either textual or graphical, and can be composed and subsequently imported from external packages. VideoNoter's particular strength is the interface to the video control functions. It is also one of the few analysis tools which allows users to easily customize their own annotation screen. Users may define their own button oriented coding template for marking events of interest and may reorder the columns by select-and-drag operations. The function of each column is user-definable. Automatic control of video functions are accessible implicitly through the worksheet and scroll bars, or explicitly through a menu bar. User have had to rely heavily on textual entry of data from a keyboard directly into columns in the data file. This time-consuming data coding has restricted the use of this system.

EVA is an interactive video annotation tool which allows both "on-line real-time" coding while the experiment is running, and "off-line" detailed coding with the stored video record after the experiment is competed. (MacKay, 1989; MacKay and Davenport, 1989). It allows experimenters to enter notes, verbal transcriptions, keystroke logging for the subjects, and symbolic categories for organizing the recorded data. One interesting capability of this tool allows the text transcriptions to appear as "subtitles", synchronized with the video. One subtitle appears for each participant. Another facility, not seen in other tools, is the ability to automatically log keystrokes from the subjects and synchronize them with the video. These can be presented in a manner similar to the "transcription subtitles".

U-Test is a tool developed expressly for usability testing and is fairly representative of many usability testing systems (Kennedy, 1989). The emphasis is on "real-time on-line" coding. The tool is pre-programmed, by the experimenter, with a list of tasks that the subjects are to perform. For each task, a detailed list of steps and a set of anticipated possible errors which the subject might make are given. These "error buttons" may be considered specific cases of experimenter-created event indices used to automatically index to specific points on the video tapes. A "timer on" and "timer off" function allows experimenters to set start and stop points for intervals of interest. Experimenters may also enter text comments and observations. These are linked to the video tapes and may be used as indices to specific points on the tapes. The U-Test tool also provides the experimenter with reminders about when they must perform a certain action with regards to the experiment itself.

One of the few tools designed specifically for inexperienced users is Xerox EuroPARC's "virtual VCR" (Buxton and Moran, 1990). A graphical image of a VCR control panel is presented on the user's computer screen. This panel contains all of the standard control functions. In addition to the control functions, the users may "mark" the tape with indices and associated comments (in much the same manner as proposed earlier in this paper). Comments are restricted to short one line titles or notes. These "tags" have a designated start and stop point and a GOTO function for playback.

These systems each address different problems in video analysis. They, and the VANNA system described below, are summarized in Table 1 (shown at the end of this paper) using the functional specifications described earlier.

The VANNA System

The VANNA (Video ANNotation and Analysis) system integrates, on a Macintosh, various multimedia elements into a single video document. User interfaces for the system were created using brainstorming sessions, iterative design, and rapid prototyping, resulting in many versions of the system over a short period of time (approximately 8 versions in 4 months). We used direct manipulation interfaces to support a number of important features described later in this paper. Additionally, we designed the system to support a variety of input devices including a touch screen, digital stylus, mouse, and keyboard. The system supports both real-time annotation and the detailed analysis of video data.

Coding the Data

Users define their own index markers by duplicating index buttons and assigning each a unique name. A single button press immediately creates an index label and links it to the corresponding location in the video document. These indices are used to capture the occurrence of important events in the video and can also be labelled to reflect rankings of behavioral data such as mood and mood changes. Similarly, to capture events having durations, a special index type called an interval is used, indicated using a start/stop button or switch. Users may define any number of indices or intervals. A typical coding screen is shown as Figure 1.

Textual comments may be entered either alone or in conjunction with an index or interval. The comment window is variable length, scrollable and editable. Verbal transcriptions may be thought of as a special case of commenting and are therefore entered in a similar manner. Brief comments of less than 20 characters are typically used for real-time annotation, while lengthy paragraphs and verbal protocols are entered in the more detailed analysis stages. Currently comments may be explicitly linked to any event or interval using a special "link" button.

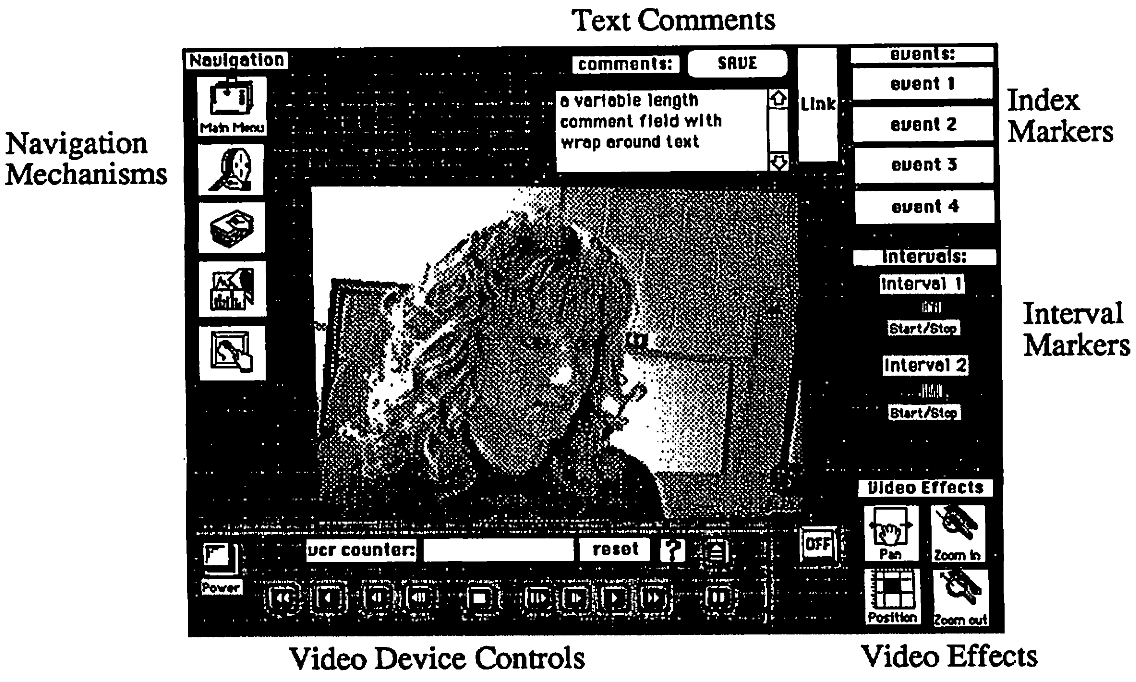
Analyzing and Interpreting the Data

All annotations are recorded in a log file, with each item type recorded in a different column (e.g., time, indices and intervals, comments). A typical VANNA log file is shown as Figure 2. The log file may be viewed, sorted by any column, edited, searched and played back. Keyword searching and sorting are provided for indices, intervals, comments and transcriptions. Items may be played back by selecting (and hence highlighting) the desired item and pressing the "play" button. Simple built-in data analysis routines calculate the frequency of occurrence for each index label and interval, the average and cumulative durations for each interval, and the variability in duration for each interval. For more detailed statistical analysis or graphical plotting, data may be exported to an external package.

User Interface and Device Control

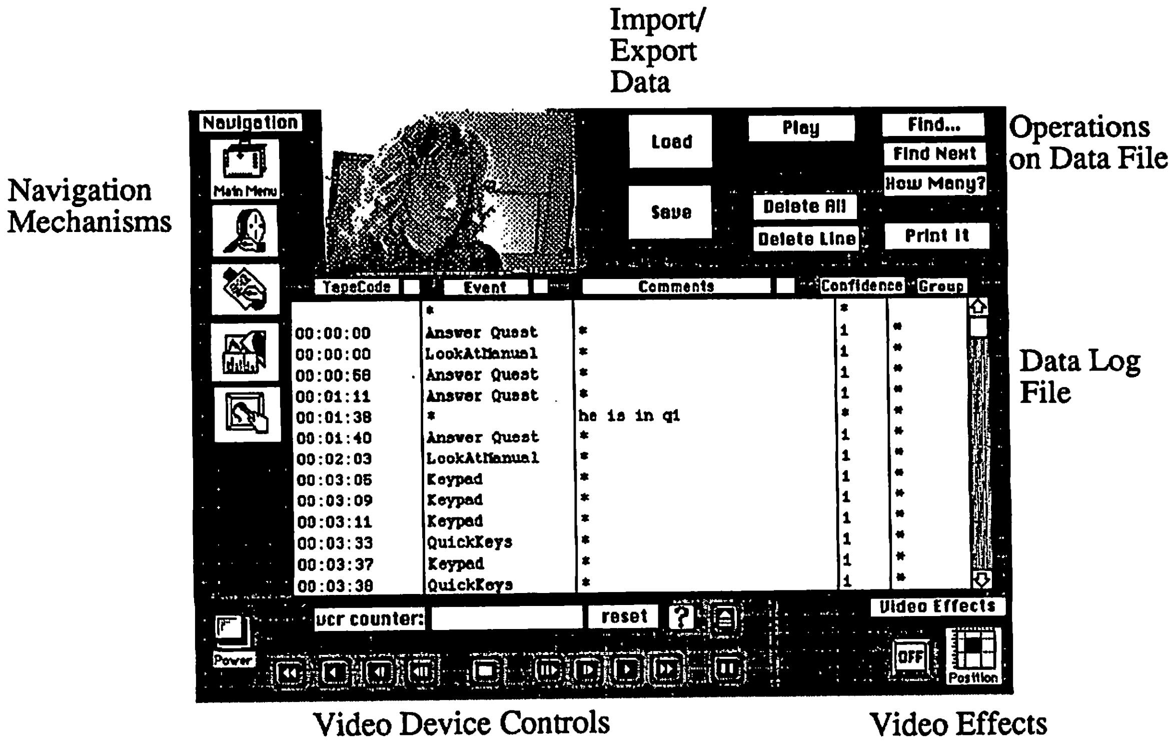
The VANNA system overview is shown as Figure 3 (at the end of this paper). VANNA simplifies technology access through software interfaces which send and receive video device commands through the Macintosh serial ports. Regardless of video device (e.g., VHS VCR, 8mm VCR, camcorder, video disc) the user sees the same iconic video button controls. Icons are based on the standard video controls found on the devices themselves.

The video image, which may be generated by a video card or by a software solution such as Apple's QuickTime®, appears as a window in the computer monitor. Users may magnify the any portion of the image (zoom in), may shift views of the image (pan), or may pull back from the image (zoom out). This results in a complete integration of video, audio and computer tools, solving many of the problems outlined earlier. The video device(s) may reside in a



a variable length comment field with wrap around text

Figure 1. Sample Coding Screen



TapeCode	Event	Comments	Confidence	Group
	*		*	
00:00:00	Answer Quest	*	1	*
00:00:00	LookAtManual	*	1	*
00:00:58	Answer Quest	*	1	*
00:01:11	Answer Quest	*	1	*
00:01:38	*	he is in q1	*	*
00:01:40	Answer Quest	*	1	*
00:02:03	LookAtManual	*	1	*
00:03:05	Keypad	*	1	*
00:03:09	Keypad	*	1	*
00:03:11	Keypad	*	1	*
00:03:33	QuickKeys	*	1	*
00:03:37	Keypad	*	1	*
00:03:38	QuickKeys	*	1	*

Figure 2. Sample Data Log File

different room and act as a server(s) to many users. In fact, multiple users may alternately control the same device when working collaboratively. This integration also minimizes work space size requirements (i.e., only the computer workstation is required).

Once created, selected items or groups of items can be played back automatically. (Items may be intervals, indices, comments, or graphics.) The annotation system finds the appropriate location in the video document and begins the playback sequence. This provides users with the ability to easily create "edit lists" of video segments for presentation, based on a number of user defined criteria.

The interface for entering annotations has been designed to reduce both perceptual and cognitive load. VANNA provides users with several default templates or screen layouts and a dictionary of functions. Users may add, delete, resize or relocate any object on a template, including the video window, by directly manipulating the objects themselves (e.g., cut, copy, paste, drag). Only the functions deemed necessary by the user are presented. This creates a completely customizable system.

Multiple input devices are supported simultaneously (touch screen, stylus, mouse, keyboard, video shuttle speed dial). The users may rapidly switch between input devices as appropriate. A shuttle speed dial is available for controlling the video direction and speed. This allows users to take advantage of two-handed input techniques and parallel manipulation strategies (Buxton and Myers, 1986) by controlling the video speed with one hand and pressing buttons with the other. This is particularly useful for detailed analyses when reviewing one segment of video many times at varying speeds to capture information. These devices, combined with the user definable screen layout ability, support both right and left handed subjects equally well.

Any button press provides the user with both auditory and visual feedback. Buttons temporarily inverse highlight and a brief tone sounds (a "clicking" sound like a mechanical button). Different pitches distinguish indices from intervals (in addition to a different visual appearance). The graphics of the interval button changes to differentiate between open and closed intervals. Error tones are louder, with different classes of errors being distinguished by both pitch and tone. Only critical errors display messages, minimizing interference with the primary annotation task.

Displaying the Data

Currently, the system produces reports which display data items in columns. Users define the number of columns and the column content. Columns typically contain the time, index label or interval label, user comments, and verbal transcriptions. The entire contents of the data file may be printed or users may elect to filter the data and print out a pre-determined subset. Additionally, users may view or print out reports using the simple statistics and frequency counters built into the system. Built-in statistics currently include frequency of occurrence for each index and each interval, and the cumulative duration, average duration and variability in duration for each intervals.

User Testing

The VANNA system is undergoing extensive usability testing with a variety of real tasks and applications. These include user interface testing for experiments with pie menus, behavioral studies of writing strategies in joint authoring, studies of gaze patterns for video usage in collaborative programming, usability testing scenarios for several complex devices, studies of human error in kinesiology, and naturally the analysis of video analysis sessions using the VANNA system. A number of interesting results have been observed thus far.

Users rapidly adopted personalized analysis styles, with some favouring real-time annotation and frequent "loopbacks" and others favouring analysis in slow motion with few loopbacks. The button style index markers worked very well and the performance was good for both real-time and non-real-time analysis. In both the real-time and the non-real-time analysis processes users tended to enter comments which were less than 20 characters, though the comments input field was variable length to allow for much longer text items.

Users entered data in sequences of "grouped" button presses. These groupings reflected the data characteristics; some types of events frequently occur in rapid succession. Users physically layed-out the coding screens to reflect these data characteristics by clustering related buttons together. As the data characteristics changed over time, users dynamically altered the screen layout to reflect these changes.

Most users adopted an off-line data coding strategy of entering about 10 items which they then reviewed in the log file for correctness. They would immediately make changes if necessary and then return to the coding process and enter another 10 items. This process resulted in many users requesting a automatically scrolling window view of the most recently entered data as part of the coding screen. By merging a brief view with the coding screen, users felt that their revision and coding process would be simplified. This additionally provides users with feedback about what has been recorded in the data log file.

Comments could be used as data items in themselves or could be descriptors for index markers by explicitly "linking" them. This latter case required the use of a "link" button which has proved to be problematic. Users tended to forget to press this link button when they wished to associate the comment with a specific index marker. Changing the size, location, and label of the link button did not correct the difficulty. A better mechanism for achieving this functionality is needed.

Keyword searching was used extensively in the playback process especially on experimenter comments. The ability to sort by index names and then playback many items from same group was also found to be very useful and was used extensively.

The touch screen was not used extensively but this can be primarily attributed to the angle of the screen, which was found to be too tiring for long sessions. (Most sessions lasted at least 1 hour with an average time of about 2 hours). For future use the touchscreen needs to be mounted in a recessed surface at about a 35 degree angle.

Although video zoom was provided the performance was too slow. Users wanted to rapidly magnify and later de-magnify portions of the video image. Better video technologies now make this possible.

Summary and Conclusions

The VANNA system was designed by applying proven methodologies in HCI and human factors theories. It illustrates one method of achieving a cost-effective and useful desktop video annotation and analysis system. Preliminary results in user testing indicate that the VANNA system is suitable for a number of applications for a number of users. Insights from user testing have encouraged us to contemplate a number of extensions.

For example, we have recently implemented a portable version of the annotation subsystem which runs on a laptop computer, the PowerBook. Portable Vanna can be taken into the field and allows real-time annotation to occur simultaneously with data capture.

Graphical overlay capabilities are under implementation. This will allow users to draw sketches using a stylus directly over the video image. We believe that this will prove a useful mechanism for capturing non-verbal and gestural data in behavioral analyses.

VANNA will also be linked to an automatic audio tracking system (Sellen, 1992). This system separates, logs, and graphically plots over time audio contributions from up to four meeting participants. This facilitates the analysis of speech patterns such as pauses, interruptions, and simultaneous speech.

Finally, we are currently implementing graphical time line displays and are investigating the use of color displays and animation for more vivid presentations of results.

Acknowledgements

We appreciate the support to our laboratory and to the first author from the Natural Sciences and Research Council of Canada, the Information Technology Research Centre of Excellence of Ontario, the Institute for Robotics and Intelligent Systems of Canada, Bell-Northern Research, Apple Computer, and Xerox Corporation.

References

Anacona, B. (Ed.) (1974). *The Video Handbook.* New York: Media Horizons.

Baecker, R. M. (Ed.) (1992). *Readings in Groupware and Computer Supported Cooperative Work.* Morgan Kaufmann Publishers.

Bales, R.F., (1950). *Interaction Process Analysis: A Method for the Study of Small Groups.* Addison-Wesley.

Bales, R.F. and Cohen, S.P. (1979). *SYMLOG: A System for the Multiple Level Observation of Groups.* Free Press.

Buxton, W. and Moran, T. (1990). EuroPARC's Integrated Interactive Intermedia Facility (IIIF): Early Experiences. In *Multi-user Interfaces and Applications.* S.Gibbs and A.A. Verrijn-Stuart (Eds). North-Holland. 11-34.

Buxton, W. and Myers, B. (1986). A Study in Two-Handed Input. In *Proceedings of ACM SIGCHI '86.* 321-326.

Dranov, P., Moore, L. and Hickey, A. (1980). *Video in the 80's: Emerging Uses for Television in Business, Education, Medicine and Government.* White Plains, NY: Knowledge Industry Publications.

Greif, I. (Ed.) (1988). *Computer Supported Cooperative Work: A Book of Readings.* Morgan Kaufmann Publishers.

Harrison. B. L. (1991). The Annotation and Analysis of Video Documents. M.A.Sc. Thesis, Dept. of Industrial Engineering, University of Toronto. April 1991.

Harrison, B.L. and Baecker, R.M. (1991). Video Analysis in Collaborative Work. Working paper. Dynamic Graphics Project, Computer Systems Research Institute, University of Toronto.

Heath, C. (1986). *Body Movement and Speech in Medical Interaction.* Cambridge, England: Cambridge University Press.

Heath, C. (1990). Virtual Looking. Rank Xerox EuroPARC, working paper.

Hutchinson, A. (1954). *Labanotation.* New York: New Directions Publishers.

Ishii, H. (1990). TeamWorkStation: Towards a Seamless Shared Workspace. *Proceedings of the ACM CSCW'90 Conference on Computer-Supported Cooperative Work.*

Kennedy, S. (1989). Using Video in the BNR Usability Lab. *SIGCHI Bulletin* 21(2), October 1989, 92-95.

Laban, R. (1956). *Principles of Dance and Movement Notation..* London: Macdonald and Evans.

Losada, M. and Markovitch, S., (1990). GroupAnalyzer: A System for Dynamic Analysis of Group Interaction, *Proceedings of the 23rd Annual Hawaii International Conference on System Sciences,* IEEE Computer Society, 101-110.

Losada, M., Sanchez, P. and Noble, E.E., (1990). Collaborative Technology and Group Process Feedback: Their Impact on Interactive Sequences in Meetings, *Proceedings of the ACM CSCW'90 Conference on Computer-Supported Cooperative Work..*

MacKay, W.E. (1989). EVA: An Experimental Video Annotator for Symbolic Analysis of Video Data, *SIGCHI Bulletin* 21(2), October 1989, 68-71.

MacKay, W.E. and Davenport, G. (1989). Virtual Video Editing in Interactive Multimedia Applications. *Communications of the ACM,* 32(7), 802-810.

Mantei, M., Baecker, R. M., Sellen, A. J., Buxton, W., Milligan, T., and Wellman, B. (1991). Experiences in the Use of a Media Space. *Proceedings of ACM SIGCHI '91.* 203-208.

Nielsen, J. (1990). Big Paybacks from "Discount" Usability Engineering. *IEEE Software*, 7(3), May 1990. 107-108.

Olson, J.S. and Storrosten, M. (1990). Finding the Golden Thread: Representations for the Analysis of Videotaped Group Work. University of Michigan, working paper.

Potel, M.J. and Sayre, R.E., (1976). Interacting with the Galatea Film Analysis System. ACM Computer Graphics 10(2), July 1976, 52-59.

Potel, M.J., Sayre, R.E. and MacKay, S.A., (1980). Graphics Input Tools for Interactive Motion Analysis. *Computer Graphics and Image Processing* 14, 1-23.

Ramey, J. (1989). A Selected Bibliography: A Beginner's Guide to Usability Testing, *IEEE Transactions on Professional Communication*, 32(4), December 1989.

Rein, G. (May, 1990). A Group Mood Meter. *MCC Technical Report.*

Roschelle, J., Pea, R. and Trigg, R. (1990). VideoNoter: A Tool for Exploratory Video Analysis. *IRL Technical Report* No. IRL90-0021, March 1990.

Sellen, A. J. (1992). Speech Patterns in Video-Mediated Conversations. *Proceedings of ACM SIGCHI '92,* to appear.

Trigg, R. (1989). Computer Support for Transcribing Recorded Activity. *SIGCHI Bulletin* 21(2), October 1989.

Functional specifications	Galatea	GroupAnal.	VideoNoter	EVA	U-Test	Virt. VCR	VANNA
1. User-specified indices	+	+	+	+	+	+	++
2. Grouping items					+		++
3. Experimenter comments	+	++	++		+	+	+
4. Verbal transcription		+	+	+	+		*
5. Characteristics measures		++				+	
6. Non-verbal information	+		+	+			*
7. Keyword searching		+	+	++	++	+	++
8. Keystroke integration			+	++	++		+
9. Import/export data	+	+	++	++			+
10. Interaction patterns		++					
11. Interjudge reliability		++	+	++	+		
12. Video controls	++	+	++	+	+	++	++
13. Previous/next item	++	+	++	+	+	+	++
14. Sets of items			?	?	+	+	+
15. Direct manip. interface	+	+	+	+	+	+	++
16. Simple mental models	++	+	++	+	+	++	++
17. Customizable screens	++		++	+	++		++
18. Multi-media document	++	+	+	++	+	+	++
19. Synchronizing tapes				+	+		
20. Customizable presentation	+	++	++	+	+		+
21. Time line displays		++	+	?			*
22. Anim/colour displays	+	++					*
23. Video presentations	+	+	+	++	++	++	++

Table 1. Assessment of Video Analysis Tools

Legend:	++	Superior capability	*	Planned but not yet implemented
	+	Basic capability	?	Information not available
		Capability not in system (Blank entry)		

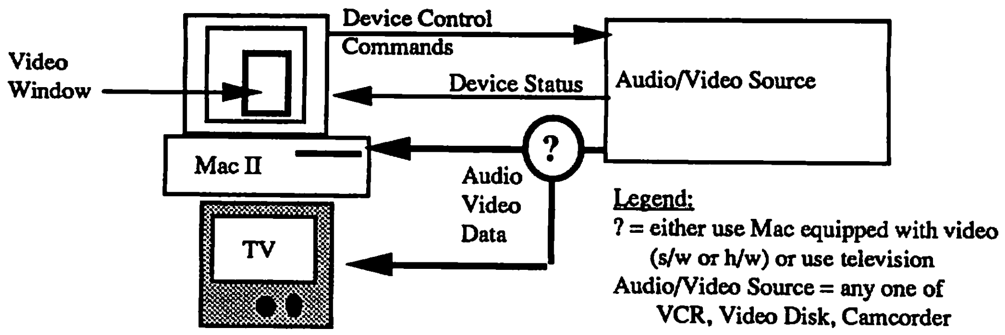

Video Window
Device Control Commands
Device Status
Audio/Video Source
Mac II
?
Audio Video Data
TV
Legend:
? = either use Mac equipped with video (s/w or h/w) or use television
Audio/Video Source = any one of VCR, Video Disk, Camcorder

Figure 3. VANNA system Overview

A Data Parallel Algorithm for Raytracing of Heterogeneous Databases

Peter Schröder[*] **Steven M. Drucker**[†]
Thinking Machines Corporation
245 First St.
Cambridge, MA 02142-1214

Abstract

We describe a new data parallel algorithm for raytracing. Load balancing is achieved through the use of *processor allocation*, which continually remaps available resources. In this manner heterogeneous data bases are handled without the usual problems of low resource usage. The proposed approach adapts well to both extremes: a small number of rays and a large database; a large number of rays and a small database. The algorithm scales linearly—over a wide range—in the number of rays and available processors. We present an implementation on the Connection Machine CM2 system and provide timings.

Résumé

Cet article présente un nouvel algorithme parallèle pour le lancer de rayons. L'*allocation des processeurs*, qui distribue la tâche aux resources disponibles, permet de garder une charge bien répartie. Ainsi évitons nous les problèmes usuels dus aux ressources de bas niveau tout en manipulant des structures de données hétérogènes. Notre approche s'applique aussi bien à un faible nombre de rayons et des données importantes qu'à un nombre important de lancer de rayons sur des données de taille limitée. L'algorithme fonctionne en temps linéaire en le nombre de rayons et de processeurs disponibles. Nous présentons une implémentation sur le système Connection Machine CM2 et donnons des temps de calcul.

Keywords: Massively parallel, SIMD, raytracing.

Introduction

With the advent of massively parallel computers, many algorithms which require large computing resources have been developed to take advantage of these new machines [2; 20]. While the use of parallel computers, like the Connection Machine CM2 system, has become almost common place in computational fluid dynamics and QCD theories [6], only relatively few graphics algorithms for general purpose parallel computers have been published. We hypothesize that this is due in large measure to the extraordinary success and availability of cheap special purpose graphics hardware.

Most of the work on this hardware (see for example [14; 25; 1; 22; 27]) has focused on polygon scanline rendering. At the same time, the development in rendering algorithms has been towards more sophisticated global shading models. Unfortunately, these can only make limited use of commonly available graphics hardware (e.g. the use of z-buffers for radiosity and raytracing acceleration [9; 31]). There has also been some interest in the use of distributed processing and multi-processing, to accelerate radiosity computations [28; 3], and ray tracing [16; 26; 12]. As the computational needs of sophisticated global illumination models continue to increase and the size of data sets continues to grow (e.g. volume rendering) the importance of algorithms for scalable, general purpose computers will continue to increase.

Applications such as volume rendering, ray tracing, and radiosity have computational needs large enough to take advantage of highly parallel architectures. While, for example, ray tracing appears to parallelize trivially, the published research shows otherwise (see below). Many of the published algorithms struggle with the load balancing issue which becomes worse with increasing numbers of processors. There are also examples—from volume raytracing—where two different algorithms, which are equivalent on serial machines, can have radically different run times on parallel architectures [30]. The Achilles heel of most graphics algorithms on parallel architectures is their high demand for general, versus regular, communication. As the following review of previous work shows, the main focus in this area of research is on minimizing general communication.

In the next section we discuss in detail previous work in parallel raytracing. Following that we describe our new algorithm and present timing results, finishing off with conclusions. The details of the implementation are described in the appendix.

Previous work in parallel ray tracing

Most previous work concerns the mapping of ray tracing onto MIMD hypercubes. Carter and Teague [8] discuss a simple scheme of replicating the entire database at every

[*]Author's current address: Department of Computer Science, 35 Olden St., Princeton University, Princeton, NJ 08544-2087, e-mail: ps@cs.princeton.edu

[†]Author's current address: Media Laboratory, 20 Ames St., Massachusetts Institute of Technology, Cambridge, MA 02142, e-mail: smd@media.mit.edu

node of an iPSC/2 computer. The speedup factor gained by the use of many processors is almost exactly the number of processors in the system. The only issue is the assignment of pixels to processors. For this Carter and Teague use the obvious "pixel mod number of processors" *comb* assignment (this same approach was also chosen by [24] for a raytracer on the Pixel machine). Salmon and Goldsmith [29] consider this scheme as well. They go on to consider the more general case of databases that are too large to be replicated at every node. They use a bounding hierarchy data structure for their database and replicate only the top n levels of this structure at every node. In this way the more frequent intersection tests against the top of the hierarchy can be performed locally. The sub trees of level $k > n$ are distributed across the nodes. Considering the statistics of intersection of rays against subtree nodes of the bounding hierarchy, they develop a procedure to determine the height of the subtrees which are deposited across processors.

Another decomposition scheme using a bounding hierarchy was explored in Carter and Teague [7]. The entire bounding hierarchy tree, except for the leaf nodes, is replicated in all processors. The leaf nodes are distributed across the processors. An LRU cache is used to maintain a set of primitives at each processor. Using a blocking scheme to distribute pixels across processors they naturally take advantage of the implied spatial coherence. Load balancing is achieved by a master process which allocates pixel blocks to idle processors.

A different set of algorithms uses SIMD architectures, such as the Connection Machine CM2 system, for *data parallel* ray tracing. While some of the design issues are similar, for example keeping global communication to a minimum, others are quite different. In data parallel algorithms, there is only one thread of execution, and load balancing takes on a new meaning. Instead of distributing possibly different parts of the algorithm across processors, load balancing aims to keep the set of active processors as large as possible for every part of the algorithm.

Delany [11] considers the case of a bounding hierarchy and its efficient traversal. The tree structure is mapped onto processors using the numbering of a preorder traversal of the tree [4]. Assuming that the bounding volumes are the same as the objects to be ray traced, nodes and leaves are of the same type. As is the case, for example, in sphere databases. The algorithm assigns each ray to a processor and at every iteration uses general communication to fetch the next bounding volume as the rays traverse the tree. Using the Euler tour numbering it becomes simple for each ray to find the processor address of the next tree node to intersect with.

Delany has also described a raytracer based on a space subdivision scheme [10]. In this algorithm all objects and rays receive a tag, indicating which leaf node in a uniform octree partitioning of space they currently reside in. The least significant bit of the tag is used to differentiate between rays and objects. The algorithm proceeds by sorting rays and objects based on their tags. As a consequence, rays end up immediately adjacent (in processor space) to objects that they need to intersect with. Using the *scan communication* primitives[1] object data can be efficiently propagated to all rays which occupy the same voxel as the given object. By doing backward and forward scans on the bits of the tags in the sorted list of rays and objects, it is possible for a ray to compute the next non-empty voxel along its direction to within a power of 2. In this way rays are advanced to the next parent voxel in the octree that contains another object and the closest intersection can be found in logarithmic time in the length of the ray.

Discussion

The MIMD hypercube algorithms are generally characterized by their difficulties with load balancing. For example Carter and Teague [8] replicate the entire database at every node. This leaves the distribution of rays to even out the load. For large numbers of rays—small numbers of processors in the system—the simple *comb* assignment does well. As the number of rays per processor decreases, though, the load imbalance goes up markedly. Furthermore we do not consider the replication of the entire database at every node feasible as this limits the size of databases and leads to ever increasing waste of memory as the number of processors increases. Salmon and Goldsmith [29] distribute their database across the processors and use a static analysis to determine how best to achieve this. The actual performance numbers they give indicate even higher load imbalance for increasing numbers of processors (almost 40%). This is not surprising since the work estimates, on which the decomposition is based, exhibit large variance for small numbers of rays per processor; as would be the case with increasing numbers of processors. These concerns also apply to the different distribution scheme of Carter and Teague in [7]. Since their distribution approach adjusts dynamically, through the use of an LRU cache, it exhibits somewhat better load balancing behavior. Both approaches assume that communication is *very* expensive and explicitly maintain multiple processes at each node to mask the idle time associated with communication. Since these algorithms exist in a MIMD message passing context they easily deal with databases containing arbitrary mixtures of object types.

The concerns of the SIMD algorithms are typically centered around the fact that all processors need to execute in lockstep. While this simplifies program development greatly, it leaves these algorithms very vulnerable to low resource usage in the face of multiple object types. In particular, Delany's algorithm [11] is even more restrictive in that the bounding objects must be of the same type as the primitive objects themselves. Some flexibility in the object types is afforded by using a class of objects for which ray/object intersections can be ex-

[1]Briefly, *scan* operations execute an operator such as *sum* across an ordered set of elements. These could be in a vector with a vector processor executing the instruction along the length of the vector, or an ordered set of processors with each element in its own processor. In the latter case the *scan* instructions execute in logarithmic time in the length of the set [4].

pressed in terms of a common meta type (e.g. quadric primitives, with planes as degenerate quadrics, etc.). In this algorithm the top of the tree is, at least initially, a communications bottleneck. During the initial steps of the algorithm all rays attempt to fetch data from the processor which holds the root node. This causes high congestion in the router and results in slow execution of the general communication step. As the algorithm progresses this problem is alleviated since the rays start distributing themselves across all nodes of the hierarchy and new rays are added at different times. Notice that in algorithms of this kind coherence can actually be disadvantageous as it implies high congestion in general communication patterns. A more serious problem with Delany's algorithm is the fact that regardless of the object types, terminal and non-terminal intersections require different treatment. Some processors, for example, may need to evaluate the shading model, while others need to execute further intersection tests with the hierarchy. Delany addresses this with the use of a finite state automaton which always goes to the state with the most processors waiting. Klietz [23] describes the use of these ideas in a framework allowing multiple object types—all of which are sub types of a common meta type—and the automatic object hierarchy construction as described by Goldsmith and Salmon [17]. Klietz reports linear scaling of the algorithm in the number of objects in the hierarchy up to approximately 10000 (on a 32k processor CM2 system), beyond which the performance of the general communication falls off markedly.

The other algorithm proposed by Delany [10] exploits the ability of the CM2 to sort very fast [5]. It however also requires a finite state automaton for load balancing the states, e.g. *need to intersect*, *need to advance*, etc. At each step the algorithm enters into that state which has the most processors waiting. For multiple object types the number of states and the potential for each state having only a few active processors in it increases, potentially resulting in low resource usage overall.

The above analysis shows that it is imperative to consider load balancing from the onset and assure that it scales with increasing numbers of processors. The goal of our work was to develop an algorithm which scales with the available resources—number of processors—as well as the problem size. In particular given that there are typically at least hundreds of thousands of rays it should be possible to take advantage of very large numbers of processors. All the MIMD algorithms discussed above assume small numbers of processors ($\ll$ 1024). While this is not explicit in the design, it can be seen from the analyses. As the number of processors goes up the load imbalance increases. The data parallel ray tracing algorithms scale much better with the number of processors, but suffer—as do the MIMD—algorithms from unwieldy load balancing procedures. Even though load balancing is addressed load imbalance can still be very high [29]. These issues are furthermore exacerbated in the SIMD case with the use of different object primitives.

In our algorithm we address the load balancing issue, as well as the issues connected with varying object types in the same database through the use of *processor allocation*. The basic observation is that even for small numbers of objects—say, a few triangles among many spheres in the same database—there are typically more rays *potentially* intersecting these few objects, than there are processors in the machine. Hence we can keep the available resources busy if we simply loop over the object types. This requires the algorithm to continually remap the resource usage. To be sure, this remapping requires general communication, but if the amount of computation between remapping steps is significant, the cost of the general communication steps can be amortized. In the performance section below we will argue that the current algorithm achieves this.

A new data parallel ray tracing algorithm

A simple and straightforward method of data parallel ray/object intersection would intersect every object with every ray in a single intersection step and then extract the closest intersection. This is not only very wasteful but also impractical since it amounts to computing the full cross product of all rays and objects. For a database of 2^{13} objects, and 256x256 pixels/rays we would require 2^{29} processors each executing one ray/object intersection in parallel. The algorithm we propose takes advantage of the fact that this cross product can be made sparse by observing that most rays can only *potentially* intersect a small fraction of the entire object database. The raytracing literature is full of algorithms which use, for example, bounding hierarchies, to exploit this sparsity.

The main task then is to develop an effective strategy to derive small lists of candidate objects for each ray and only intersect each ray with the objects on the respective list. Ideally this list contains only those objects which are actually along the ray. In order to find the list of candidate objects for every ray we use a coarse space subdivision of the object database. Specifically we divide the world into equal sized voxels which are coordinate axes aligned. In a preprocessing phase [15], we find all objects that overlap a given voxel and maintain a list of these for each voxel. During the ray casting step all voxels that a given ray pierces are enumerated and used to access the per-voxel lists of objects. All objects retrieved in this way are candidates and are intersected in parallel with the given ray. A *downward min scan* on the computed intersection distance gives the closest object intersected by a given ray.

Our technique of generating candidate sets is not unlike the shaft culling technique introduced recently by Haines and Wallace [19]. They consider ray casting in the context of radiosity form factor estimation. Candidate lists are comprised of all objects overlapping the the convex hull of an origin and destination object. This technique requires that for each ray an origin and destination object is known, as is the case in radiosity. For ray tracing this does not generally hold. Our technique, while coarser, does not depend on knowing such objects and thus also works for general ray tracing. Since we only use flat lists for each voxel our case corresponds to their *open always* strategy.

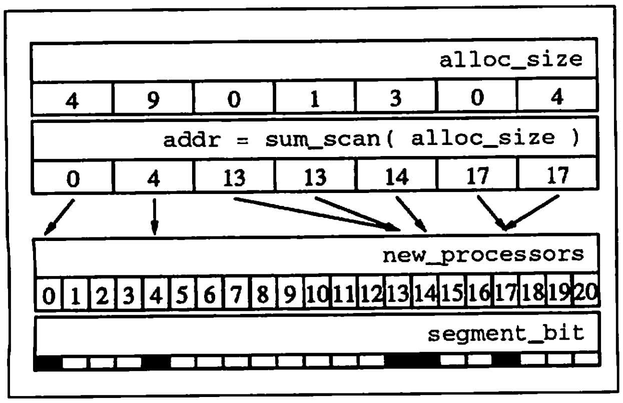


Figure 1: Each box corresponds to a processor in a 1 dimensional processor set. The parallel variable (pvar) **alloc_size** holds the number of processors to allocate. The address of each allocated segment is given by the *sum-scan* of **alloc_size**. The **segment_bit** pvar delineates the actual segments.

The details of the SIMD implementation of the algorithm can be found in Appendix . However, since processor allocation is at the heart of most steps in the algorithm we will give a brief description of this data parallel programming idiom (see also [4]). It is through the use of processor allocation that we achieve the desired load balancing.

Processor allocation

Just as serial algorithms use memory allocation to manage dynamically changing demands, data parallel algorithms use processor allocation for the same ends. Consider, for example, a ray which is to be intersected against a candidate list of objects. Each ray, which is stored in its own processor, typically needs to be intersected against different numbers of candidate objects. In order to exploit ray/object parallelism each ray-processor allocates a number of object-processors. This is accomplished by allocating a new processor set with enough processors to hold the sum total of requested processors. This new processor set is segmented so that each segment consists of as many processors as the associated requesting processor required (see figure 1). The allocating processors receive a pointer (processor address) to the segment allocated to them, which can be used to move data between the allocating and allocated processors. The **segment_bit** can be used in *segmented-scan* operations to execute instructions on a per segment basis. For example, propagating (*segmented-copy-scan*) ray data to all objects which need to be intersected with the given ray. Another typical use in our framework is the *segmented-downward-min-scan* which finds the minimum element of a given pvar on a per-segment basis. The result of this scan is left in the first processor of each segment, whence it can be retrieved easily.

The processor allocation paradigm provides a general way to implement algorithms which dynamically require new resources of uneven length. Another attendant advantage of this approach is the implied load balancing. Since each processor allocates as many new processors as it needs there are no idle processors in the new processor set [2].

Load balancing

When ray tracing in parallel, load balancing is important in two places. First the amount of work necessary to intersect a given ray with the database varies from ray to ray and second the recursion depth of a given ray tree varies based on geometry—rays leaving the scene—and object properties—highly specular versus purely diffuse objects. We use processor allocation as described above to address both of these requirements. For a given set of rays we allocate enough processors per ray to hold the candidate objects from the voxels along the ray. In this way rays that, for example, leave the database will consume less resources than those which are going through an area with many objects. Similarly for each recursion level only those rays which need to be followed further allocate child rays.

This approach is only feasible however, if the router performance is high enough to support the continuous remapping of resources during allocation. On vector computers this corresponds to their ability to perform scatter/gather operations efficiently.

Performance

We use processor allocation extensively throughout the algorithm. Since this involves general communication we need to be sensitive to the underlying communication patterns. Semi-regularity in the routing pattern can lead to very high congestion rates since some form of regularity often implies that many messages need to go through only a few nodes in the communication network. In practice, it has been observed that some of the slowest general communication patterns are very regular, while random patterns tend to yield high throughput. In our case this implies that coherence, which is usually welcome and exploited, can be very disadvantageous. The loading of scene descriptions illustrates this point well. When reading in a database the ordering in the file often correlates with spatial coherence in the database. We have found in those cases that placing the objects into a linear processor set in the order read in, yielded a running time higher then when assigning objects to bit reversed processor addresses. This effect depends on the actual database loaded. For the sphere flake with 7381 spheres and 1 quadrilateral (see [18]) we found the following representative times at a voxel scale of 80^3:

Mach. size	linear load	bit rev. load
16k	208.4 S	170.8 S
32k	118.7 S	98.3 S

The granularity of the database voxelization exerts a much larger influence on the runtime. Two forces need to

[2]There can of course be idle processors if the total number of processors requested is not some integer multiple of the number of physical processors.

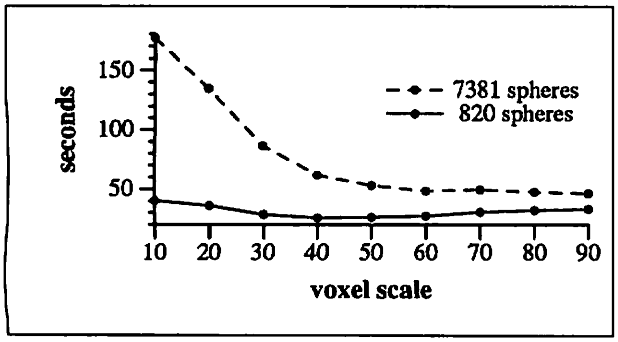


Figure 2: Timings for various voxelization scales. A bounding box for the entire data base is cut into 10^3 to 90^3 voxels. Machine size was 16k processors with an image size of 128^2 pixels.

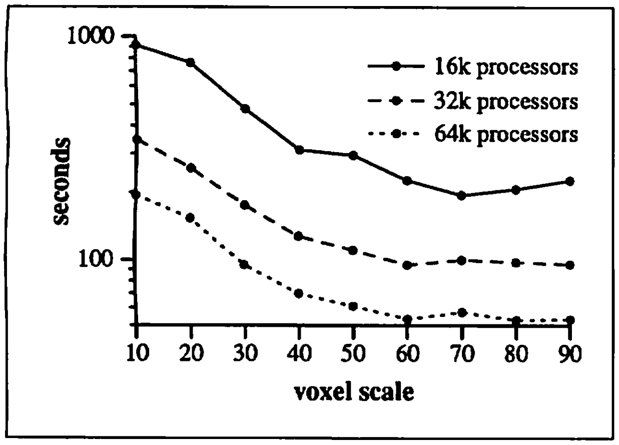


Figure 3: Timings for different machine sizes. All databases are 7381 spheres at an image size of 256^2.

be balanced here. If the voxelization of the world is very fine the candidate lists per ray are fairly small since only those objects overlapping a small neighborhood of the ray will be listed in the object fetch lists. On the other hand a fine voxelization of the world drives up the memory consumption of the system and forces us to subdivide any given ray into many and much smaller pieces. Figure 2 shows timings for two different sphere flake data bases (see [18]) with 820 and 7381 spheres respectively. These databases are good test cases since they contain both very small and very large objects. They also contain a single quadrilateral. The timings include 3 recursive reflection levels and shading via 3 light sources (shadow feelers to all light sources at all intersections). All tests were timed on a CM2 running at 7MHz.

The observed behavior with respect to voxelization scale remains for different machine sizes as well. In figure 3 we see data for the 7381 sphere database for an image size of 256^2 pixels for different machine sizes. When considering larger images with all other parameters the same the runtime scales linearly in the number of rays.

Considerable speedups can be gained from subdividing coarsely along the view direction and finely in the viewing plane. The following table gives one such example for the same scene as above on a 32k processor machine at an image resolution of 256^2

Voxel scale	Time in S	Voxel scale	Time in S
$160^2 \times 30$	95.6	$180^2 \times 20$	101.6
$160^2 \times 20$	90.6	$160^2 \times 20$	90.6
$160^2 \times 10$	97.7	$140^2 \times 20$	95.7

Using our algorithm on various scenes we have found that there always existed an optimal (in the sense of runtime) subdivision granularity. Due to memory constraints this subdivision can not always be attained. Since in most scenes the large majority of rays are view rays it is advisable to transform the database such that the view plane is orthogonal to one of the coordinate axes. By subdividing along that coordinate axis coarsely the incidence of a ray being intersected multiple times with the same object is reduced. At the same time a tight fit—and a small candidate object list—is achieved through the fine subdivision in the image plane. This holds across different machine sizes as well (see figure 4).

Conclusions and future work

We have developed a data parallel ray/object intersection algorithm which scales in the number of processors as well as in the number of rays and objects over several binary orders of magnitude. Through the use of processor allocation every stage of the algorithm is efficiently mapped onto the available resources without requiring explicit load balancing steps. Since the algorithm uses a large amount of general communication, parameters such as subdivision size and ordering of objects in processor space need to be tuned to achieve maximum performance. In particular we observe that the voxelization should be coarse along the view rays and fine parallel to the image plane. The algorithm has been incorporated successfully into a massively parallel radiosity algorithm [13].

In the current implementation the performance is limited, since we do not take advantage of any coherence along the

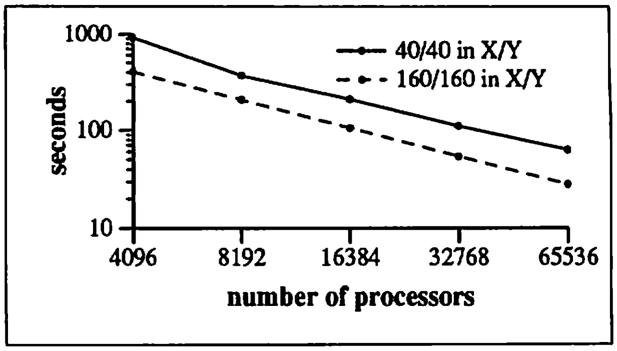


Figure 4: Runtimes for different, weighted voxelization scales. The image size is 256^2 and subdivision in Z is 20 for all runs while the subdivisions in X and Y are 40/40 and 160/160 respectively. The database was the same as in the other examples.

ray. When an object overlaps several voxels along a given ray, the ray will be intersected with it multiple times. Many unnecessary intersection tests also occur because rays are intersected in parallel with all the objects along the ray's length. This implies that there is no notion of stopping as soon as the first intersection is found. Customary acceleration techniques like shadow hit caches, are not currently incorporated into our approach. These shortcomings cannot trivially be addressed in a strict SIMD framework as it is provided by the CM2 Connection Machine System but would be straightforward in a data parallel MIMD framework as given by the CM5 computer, or other MIMD machines. In particular the intersection tests should progress through the candidate lists of each individual voxel in parametric order along the ray and finish as soon as an intersection is found. The problem of a ray intersecting itself multiple times with the same object is not as easy to address. The usual technique of depositing a tag with the object consumes too much memory in our case since all rays are traced in parallel and a possibly very large number of rays can attempt to deposit a tag.

The absolute performance of the current implementation is comparable to the best algorithms on the fastest workstations. This is mostly due to the lack of optimizations in our current implementation. As mentioned above some of these could be added, while others would require the more flexible framework of a MIMD computer. The algorithm also requires a large amount of general communication during load balancing. Current communications networks do not provide enough bandwidth to keep this part of the algorithm from consuming disproportional amounts of time. We expect this to change as communication networks become faster relative to CPU speed. The main feature of our algorithm remains its ability to scale over a wide range of resources and demands.

Acknowledgments

The authors would like to thank Thinking Machines for providing the framework and resources for this work. In particular we would like to acknowledge Lew Tucker, Gary Oberbrunner, Matt Fitzgibbon, Karl Sims, and Jim Salem for many discussions about algorithm design for massively parallel SIMD architectures. Some of the timings in this paper were made on the Los Alamos Advanced Computing Laboratory 64k Connection Machine CM2 computer with the help of Robert Kares.

References

[1] AKELY, K., AND JERMOLUK, T. High-Performance Polygon Rendering. *Computer Graphics 22*, 4 (August 1988), 239–246.

[2] BAILEY, J. Implementing Fine-grained Scientific Algorithms on the Connection Machine Supercomputer. Technical Report TR89-1, Thinking Machines Corporation, Cambridge, MA 02142, USA, 1990.

[3] BAUM, D. R., AND WINGET, J. M. Real Time Radiosity through Parallel Processing and Hardware Acceleration. *Computer Graphics 24*, 2 (March 1990), 67–75.

[4] BLELLOCH, G. *Vector Models for Data Parallel Computing.* Artificial Intelligence Series. MIT Press, Cambridge, MA, 1990.

[5] BLELLOCH, G. E., LEISERSON, C. E., MAGGS, B. M., PLAXTON, C. G., SMITH, S. J., AND ZAGHA, M. A Comparison of Sorting Algorithms for the Connection Machine CM-2. In *Proceedings Symposium on Parallel Algorithms and Architectures* (Hilton Head, SC, July 1991), pp. 3–16.

[6] BOGHOSIAN, B. M. Computational Physics on the Connection Machine. *Computers in Physics 4*, 1 (January/February 1990), 14–33.

[7] CARTER, M. B., AND TEAGUE, K. A. Distributed Object Database Ray Tracing on the Intel iPSC/2 Hypercube. In *Proceedings of the 5th Distributed Memory Computing Conference* (1990), D. W. Walker and Q. F. Stout, Eds., vol. 1, IEEE, pp. 217–222.

[8] CARTER, M. B., AND TEAGUE, K. A. The Hypercube Ray Tracer. In *Proceedings of the 5th Distributed Memory Computing Conference* (1990), D. W. Walker and Q. F. Stout, Eds., vol. 1, IEEE, pp. 212–216.

[9] COHEN, M. F., AND GREENBERG, D. P. The Hemi Cube: A Radiosity Solution for Complex Environments. *Computer Graphics 19*, 3 (July 1985), 31–40.

[10] DELANY, H. C. Ray Tracing on a Connection Machine. In *Proceedings of the 1988 ACM/INRIA International Conference on Supercomputing* (July 1988), pp. 659–667.

[11] DELANY, H. C. A Simple Hierarchical Ray Tracing Program for the Connection Machine System. Tech. Rep. VZ 88-4, Thinking Machines Corporation, Cambridge, MA, December 1988.

[12] DIPPE, M., AND SWENSEN, J. An Adaptive Subdivision Algorithm and Parallel Architecture for Realistic Image Synthesis. *Computer Graphics 18*, 3 (July 1984), 149–158.

[13] DRUCKER, S. M., AND SCHRÖDER, P. A Data Parallel Algorithm for Radiosity. In *Proceedings of Third Eurographics Workshop on Rendering* (May 1992), Eurographics. to appear.

[14] FUCHS, H., POULTON, J., EYLES, J., GREER, T., GOLDFEATHER, J., ELLSWORTH, D., MOLNAR, S., TURK, G., TEBBS, B., AND ISRAEL, L. Pixel-Planes 5: A Heterogeneous Multiprocessor Graphics System Using Processor-Enhanced Memories. *Computer Graphics 23*, 3 (July 1989), 79–88.

[15] FUJIMOTO, A., TANAKA, T., AND IWATA, K. ARTS: Accelerated Ray-Tracing System. *IEEE Computer Graphics and Applications 6*, 4 (April 1986), 16–26.

[16] GAUDET, S., HOBSON, R., CHILKA, P., AND CALVERT, T. Multiprocessor Experiments for High-

Speed Ray Tracing. *ACM Transactions on Graphics* 7, 3 (July 1988), 151–179.

[17] GOLDSMITH, J., AND SALMON, J. Automatic Creation of Object Hierarchies for Raytracing. *IEEE Computer Graphics and Applications* 7, 5 (May 1987), 14–20.

[18] HAINES, E. A Proposal for Standard Graphics Environments. *IEEE Computer Graphics and Applications* 7, 11 (November 1987), 3–5.

[19] HAINES, E. A., AND WALLACE, J. R. Shaft Culling for Efficient Ray-Traced Radiosity. In *Proceedings of Second Eurographics Workshop on Rendering* (Barcelona, Spain, May 1991), Eurographics, Springer Verlag. Also published in Siggraph 91 course notes: Frontiers of Rendering.

[20] HILLIS, D. W. *The Connection Machine.* MIT Press, 1985.

[21] KAUFMAN, A. Efficient Algorithms for 3D Scan-Conversion of Parametric Curves, Surfaces, and Volumes. *Computer Graphics 21*, 3 (July 1987), 171–179.

[22] KAUFMAN, A., AND BAKALASH, R. Memory and Processing Architecture for 3D Voxel-Based Imagery. *IEEE Computer Graphics and Applications 8*, 11 (November 1988), 10–23.

[23] KLIETZ, A. Personal communication. e-mail address: alan@msc.edu.

[24] MITCHELL, D. P. Personal communication.

[25] NADER GHARACHORLOO, E. A. Subnanosecond Pixel Rendering with Million Transistor Chips. *Computer Graphics 22*, 4 (August 1988), 41–49.

[26] NISHIMURA, H., OHNO, H., KAWATA, T., SHIRAKAWA, I., AND OMURA, K. LINKS-1: A Parallel Pipelined Multimicrocomputer System for Image Creation. In *Proceedings of the 10th Symposium on Computer Architecture* (New York, 1983), ACM, pp. 387–394.

[27] POTMESIL, M., AND HOFFERT, E. M. The Pixel Machine: A Parallel Image Computer. *Computer Graphics 23*, 3 (July 1989), 69–78.

[28] RECKER, R. J., GEORGE, D. W., AND GREENBERG, D. P. Acceleration Techniques for Progressive Refinement Radiosity. *Computer Graphics 24*, 2 (March 1990), 59–66.

[29] SALMON, J., AND GOLDSMITH, J. A Hypercube Raytracer. In *Proceedings of HCCA3* (March 1988), pp. 1194–1206.

[30] SCHRÖDER, P., AND SALEM, J. B. Fast Rotation of Volume Data on Data Parallel Architectures. In *Proceedings of Visualization 91* (October 1991), IEEE, IEEE Computer Society Press, pp. 50–57.

[31] WEGHORST, H., HOOPER, G., AND GREENBERG, D. P. Improved Computational Methods for Ray Tracing. *ACM Transaction on Graphics 3*, 1 (January 1984), 52–59.

Implementation details

Building the voxel overlap lists

When a database is initially loaded into the system the first step is to build the lists which, for a given voxel, hold all the names of objects intersecting that voxel. For each object type in turn each object allocates enough processors to hold all the voxels that its bounding box overlaps (see figure 5). Call these **cand_voxels**. Using

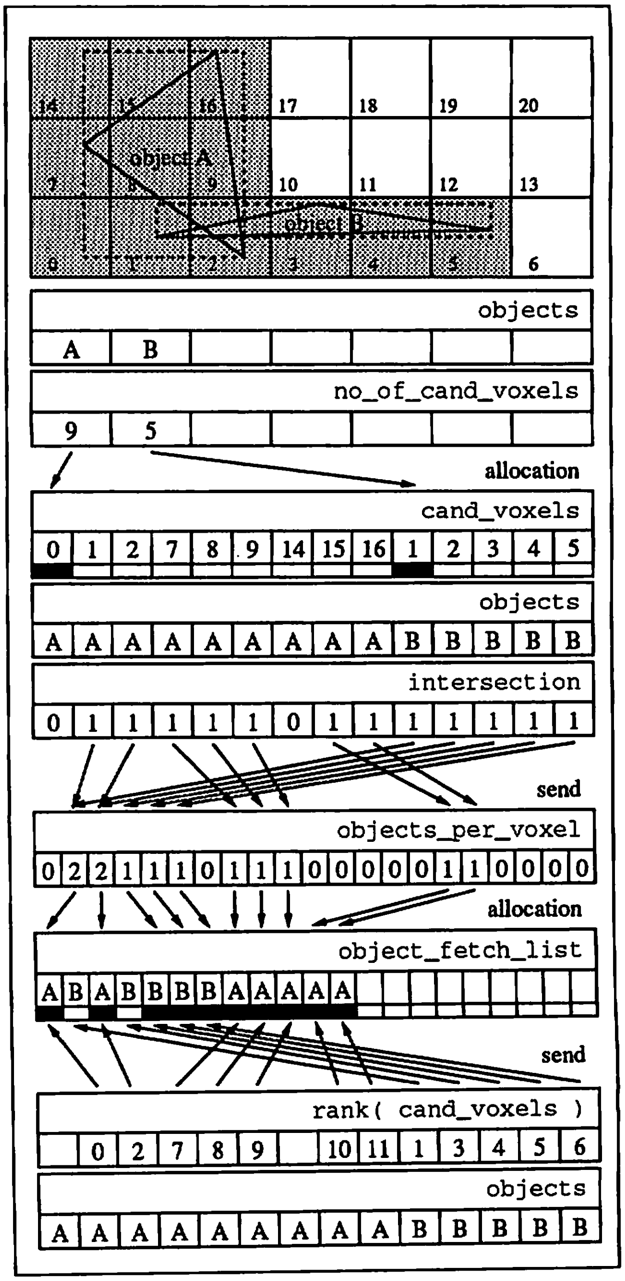


Figure 5: A 2D example of objects on a voxel grid and the voxels their bounding boxes overlap. These voxels are tested against the objects themselves. Only those that actually do intersect with the object, generate an entry in the **object_fetch_list**.

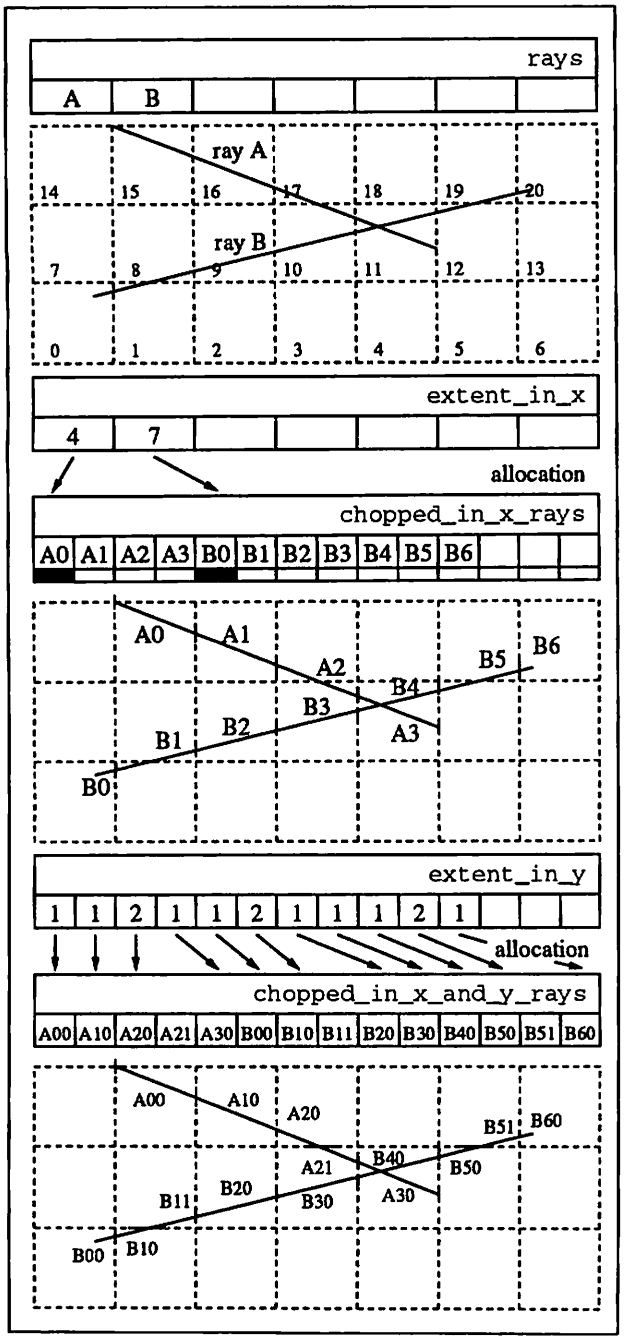


Figure 6: A 2D example of "chopping" lines to the voxel grid along the two dimensions in turn.

the forward pointers, the object data is sent to the beginning processor in each candidate segment. With a *segmented-copy-scan* this data can be propagated to every processor in its segment. At this point we properly intersect the candidate voxels with the respective object. Every processor that finds an actual intersection sends a "1" to the `objects_per_voxel` accumulator. After the intersection step each voxel holds the length of its object list in `objects_per_voxel`. In a second processor allocation step these numbers are used to allocate processors for each voxel to hold the actual lists of object ids. Call this the `object_fetch_list`. Next we find a ranking of the voxel names in `cand_voxels`. Notice that only those processors for which `intersection == TRUE` participate in this ranking. Using the rank as an address we can send the object tags to the proper places in the `object_fetch_list`, which concludes the setup. The `object_fetch_list` consists of segments, each of which corresponds to a particular voxel, that contain ids for those objects which actually intersect a given voxel. Each voxel now holds the address of the beginning of its segment in the `object_fetch_list` as well as the length of that segment. We will use these two pieces of data to build the candidate lists for each ray.

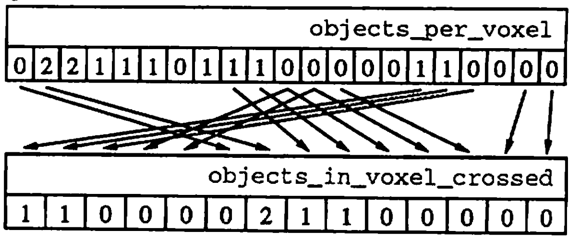
Generating ray/object candidate lists

Given a set of rays we can now build the candidate object lists. In order to do this we need to generate a list of voxels that every ray pierces. Once we have this list, it is an easy matter to use the `object_fetch_list` to get the objects themselves for the ray/object intersection test. One way to generate the voxel lists for each ray, would be to use 3D voxel conversion of the rays with 26 neighbor connectivity (see [21]). This method however is not well suited since the rays are in general of different lengths, leaving many processors idle, while the longest rays are still being voxel converted. Instead we use a nested set of 3 processor allocations, in turn "chopping" the rays to voxel boundaries along each of the 3 axes (see figure 6 for the 2D case). Notice that all children of rays A and B are contiguous and ordered along the ray in the final pvar (`chopped_in_x_and_y_rays`). When this step is completed each chopped ray simply considers its endpoints to find the address of the voxel it crosses

chopped_in_x_and_y_rays													
A00	A10	A20	A21	A30	B00	B10	B11	B20	B30	B40	B50	B51	B60

voxels_crossed													
15	16	17	10	11	0	1	8	9	10	11	12	19	20

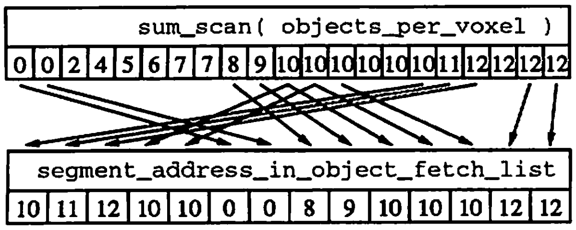
Using the address in `voxels_crossed` we can now retrieve the number of objects per voxel into `objects_in_voxel_crossed`



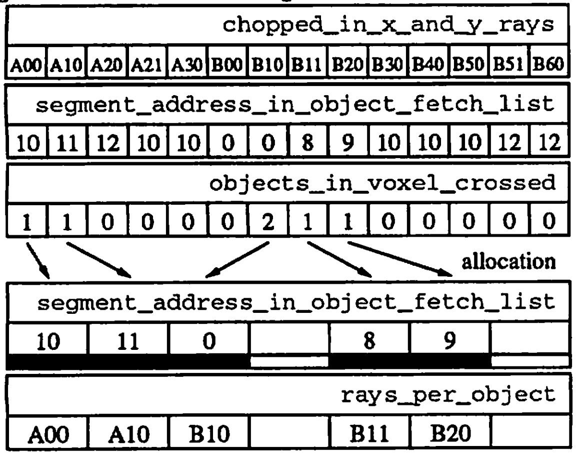
objects_per_voxel																				
0	2	2	1	1	1	0	1	1	1	0	0	0	0	0	1	1	0	0	0	0

objects_in_voxel_crossed													
1	1	0	0	0	0	2	1	1	0	0	0	0	0

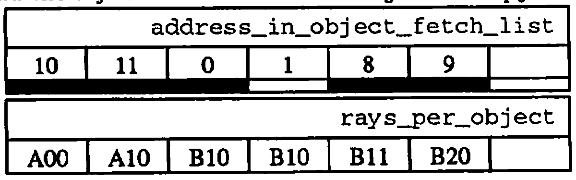
and also where these can be found in the `object_fetch_list`



objects_in_voxel_crossed indicates how many intersection processors need to be allocated. After this allocation, the segment addresses and the ray names are propagated into the allocated segments

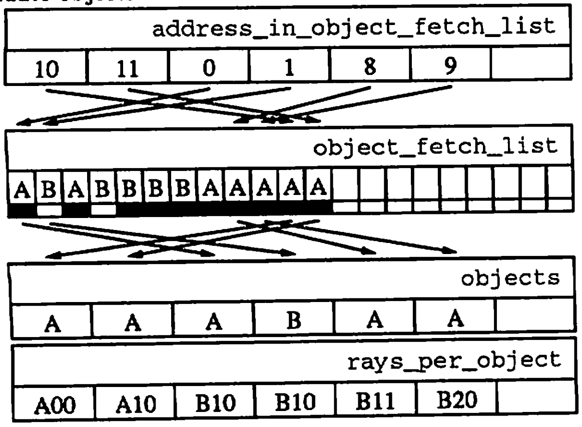


Using the segment bit the addresses of all the objects in a given segment can be generated by incrementing the segment addresses through the length of each segment and the ray data is filled in with a *segmented-copy-scan*

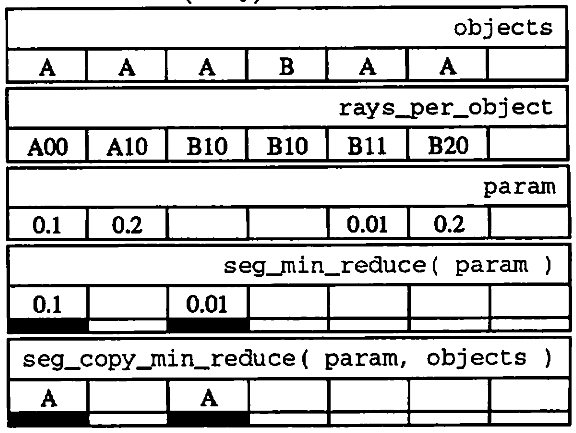


Ray/object intersection

With all the variables from the generation of the candidate list in hand we now only need an indirection through the `object_fetch_list` to pair up each ray with its candidate objects



Since the computed ray/object intersections are ordered within their respective segments in the parameter value along the ray we can execute a ***downward-min-reduce*** on the computed parameter value of the actual intersections to find the closest (if any) intersection



At this point the algorithm returns with the intersected object and the parameter along the ray where the intersection occurred.

Object Space Temporal Coherence for Ray Tracing

David A. Jevans
Apple Computer, Inc.
20525 Mariani Avenue
Cupertino, California, USA 95014

(jevans@apple.com)

Abstract

A method is presented for exploiting object space temporal coherence to speed up ray tracing of animation sequences where the camera remains static. The object space is subdivided with a hierarchical voxel grid structure. Each voxel keeps a list of the rays that pass through it when the first frame of a sequence is rendered. To render a successive frame, only rays that passed through voxels in which an object has moved are retraced. The method speeds up ray tracing of a test animation sequence by nearly a factor of four.

The method is easily adapted to work with any spatial subdivision technique. The memory requirements of the method are low.

Keywords: Ray Tracing, Frame Coherence, Space Subdivision.

1 Introduction

Ray tracing [Whitted 80] is an elegant technique for synthesizing realistic images. Numerous acceleration methods have been developed to reduce the computational expense of ray tracing, including spatial subdivision [Glassner 84] [Fujimoto 86], hierarchical object extents [Rubin 80] [Kay 86], clustering and sweeping methods [Amanatides 84] [Heckbert 84] [Shinya 87], and ray coherence [Joy 86]. These methods are effective when rendering a single image, but do not make use of the temporal coherence found in animation sequences.

The traditional way to render an animation sequence is to render each frame one at a time, ignoring any object space temporal coherence that may exist between frames. Object space temporal coherence manifests itself as objects, such as floors or walls, that move slowly or remain static throughout the course of an animation.

The aim of the research presented in this paper is to take advantage of object space coherence to speed up ray tracing of animation sequences. To be useful for high quality rendering, the method must produce images that are indistinguishable from those rendered with a traditional one frame at a time approach.

2 Previous Work

Algorithms for exploiting temporal coherence operate either in image space or object space. Image space algorithms reduce the time to render an animation sequence by rendering a subset of the pixels in a frame, and estimating the value of the unrendered pixels. Due to their sampling nature, image space algorithms may generate "incorrect" frames - frames that differ from those rendered with a traditional one frame at a time approach. Object space algorithms use information about the 3D object space, and how it changes between frames, to reduce the amount of computation to render an animation sequence.

2.1 Image Space Temporal Coherence

Badt [Badt 88] proposed a method that reduces the number of rays traced during an animation by tracing the first frame of a sequence normally, then rendering successive frames by retracing only a small random sampling of pixels for each frame. If a retraced pixel's color differs from its color in the preceding frame, a flood fill algorithm that floods both in screen space and in time is used to correct its color. Flooded pixels are retraced for preceding and succeeding frames to determine their correct colors. The flood filling reduces, but does not eliminate, the possibility of incorrect pixel colors. This method requires that the object space description for every frame of the animation sequence be available at all times during the rendering.

Chapman [Chapman 90] developed another image space algorithm that traces fewer rays than Badt's method, but is potentially less accurate. The algorithm renders every kth frame of a sequence, where $k >= 1$. The pixel colors of frame n and frame $n+k$ are compared. If a pixel's color is different in the two frames, then it is retraced at frame $n+k/2$. This process is repeated recursively, resulting in a binary search that determines the frame in which the pixel's color changed. The drawback of this algorithm is that if k is chosen to be large, high frequency changes, such as those caused by fast moving objects, will be lost.

2.2 Object Space Temporal Coherence with a Moving Camera

Hubschman [Hubschman 82] presented a method for exploiting object space temporal coherence when rendering sequences where only the camera moves. The first frame of a sequence is preprocessed to determine object visibility, and successive frames are generated by determining which objects have changed their visibility status. While the technique creates "correct" images, it does not work when objects move during the animation.

2.3 4D Ray Tracing

Spacetime ray tracing [Glassner 88] accelerates ray tracing of animation sequences through the use of hierarchies of 4D bounding volumes that encompass objects as they move through space and time. Rather than building a hierarchy of 3D bounding volumes for each frame in the animation, a hierarchy of 4D bounding volumes is created once for the animation sequence. Rays are represented as a 3D direction vector and a fourth component, their position in time. Rays are traced by testing them for intersection with the 4D bounding volumes in the scene. Only objects that lie within the 4D bounding volumes that are intersected by a ray need to be tested for intersection with it.

The main source of efficiency in this algorithm is that fewer bounding volumes are created for an animation sequence than with a traditional 3D bounding volume approach. This accelerates both the creation of bounding volumes and reduces the number of ray/volume intersection tests required to render a sequence. Motion blur by jittering rays in time is facilitated since the entire animation sequence is available to the renderer at each frame.

One drawback to this approach is that it requires an entire animation sequence to be resident in memory during rendering. The method is also not amenable to a voxel-based spatial subdivision approach. Thirdly, it does not reduce the number of rays that need to be traced at each frame.

Chapman, Calvert, and Dill [Chapman 91] developed a similar algorithm for using hierarchies of bounding volumes of animated objects. The difference with their approach is that objects inside the bounding volumes represent their motions as translation and rotation vectors. The ray/object intersection calculation is extended to encompass intersection with a moving object and to compute all intersections of a ray with a moving object. These intersections are sorted by time and distance along the ray. From this information the colors for a ray are calculated for the entire sequence, and each ray is traced only once for a given sequence.

Disadvantages to this technique are the complexity of intersection calculations and that it may not be readily extensible to handle motion that cannot be represented as simple translation and rotation vectors. Furthermore, the object bounding volumes may become large if an object moves significantly during the animation sequence,

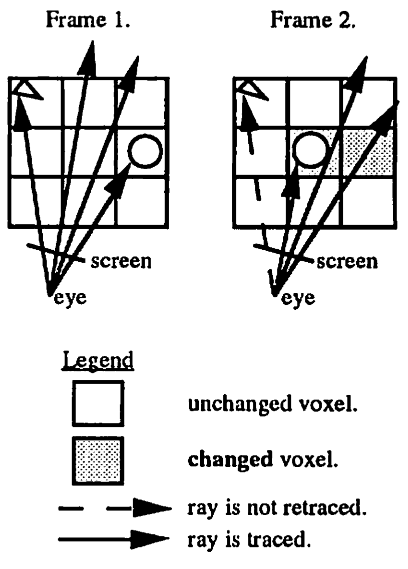


Figure 1.

reducing the effectiveness of the hierarchical bounding volume approach.

2.4 Object Space Temporal Coherence with a Static Camera

When the camera remains static during an animation sequence, the color of a pixel can change from one frame to another only if an object that is visible to the pixel has changed, or if an object has moved to become visible to the pixel. An A-buffer scan conversion renderer [Carpenter 84] can make use of this by keeping a list, throughout the animation, of the surface fragments that lie under each pixel. If an object moves between frames, it is deleted from the fragment lists of all pixels. The moved object is then rescanned into the frame buffer in its new position, and is added to the fragment lists of the pixels into which it scans. Pixels whose fragment lists have changed are then re-evaluated to determine their new color values.

Séquin presented a ray tracing algorithm that stores the ray tree at every pixel so that surface attributes of visible objects can be changed without having to retrace the image [Séquin 89]. The method fails when objects change their positions, since it cannot determine if the visible surface for a ray has changed.

Murakami and Hirota [Murakami 90] extended the algorithm to handle animated objects by subdividing the space with a voxel grid, and keeping a list of traversed voxels for every ray in a ray tree. To render a subsequent frame, all objects that move are deleted from the voxel grid, and are reinserted in their new positions. The ray trees are then examined and only rays that traversed through voxels in which an object has moved are retraced (Figure 1). A hashing scheme for representing a ray's path through the voxel space is used to speed up the process of determining which rays to retrace.

The memory requirements of the Murakami and Hirota algorithm are large, typically on the order of tens of megabytes, and grow rapidly as image resolution increases. Computational requirements also grow as a function of image resolution because the ray tree of each pixel must be examined to determine whether the ray passed through a changed voxel. Their algorithm is also specific to uniform voxel subdivision due to its use of a voxel index hashing scheme.

2.5 A New Algorithm

This paper presents an algorithm for making use of object space coherence to speed up ray tracing of animation sequences in which the camera remains static. The algorithm's memory requirements are independent of image resolution, and it is easily adapted to any spatial subdivision scheme such as uniform voxel subdivision, octree subdivision, adaptive voxel subdivision, or 5D space subdivision [Arvo 87].

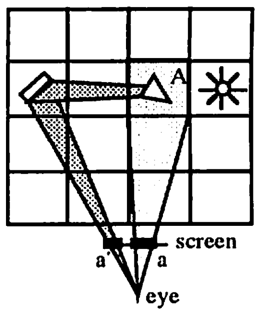
3 The Algorithm

Rays are tagged with their *x,y* pixel index in the image frame buffer. As rays are traced through a spatially subdivided scene, each voxel keeps a record of the *x,y* indices of rays that pass through it. For subsequent frames, when objects inside a voxel move, the voxel notifies the frame buffer of the pixels that will be affected. Only those pixels that are affected are retraced at each frame.

3.1 The Ray Tracer

The ray tracer used to develop this algorithm utilizes an adaptive voxel subdivision scheme [Jevans 89], although any object space subdivision scheme can be adapted to use the algorithm. The object space is subdivided by a voxel grid. Each voxel maintains a list of pointers to the objects that intersect or lie within it. If the number of objects inside a voxel is larger than some threshold, the voxel is itself subdivided with a voxel grid. A set of heuristics, based on the number of objects in a voxel, is used to determine the granularity of the subdivision [Jevans 91].

Space subdivision is done on the fly when a ray first enters a voxel. This lazy evaluation technique ensures that computation and memory are not wasted subdividing areas of the object space that are not visible. To ensure that voxels are only subdivided the first time a ray enters them, they are initially marked as *not subdivided.* When a ray enters a voxel, the heuristics are used to subdivide it, and it is marked as *subdivided*. Newly created sub-voxels are marked as *not subdivided*, as they will be considered for subdivision only if rays pass through them. Voxels marked as *subdivided* are not considered for subdivision when successive rays enter. Note that if the number of objects in a voxel is small, no subdivision may occur, but it will still be marked as *subdivided*.



Legend

☼ light source.

◁ rays directly affected by voxel A.

◀ rays affected by the shadow of an object in voxel A.

▬ a screen pixels directly affected by voxel A.

▬ a' screen pixels indirectly affected by voxel A.

Figure 2.

3.2 Rendering the First Frame

Every pixel in the first frame of the animation sequence is ray traced. Rays are labeled with their originating pixel's *x,y* frame buffer index. When a ray passes through a voxel, a record of its pixel index is stored with the voxel. This information is stored for all voxels, whether they are empty or not, and whether they are leaf or interior nodes of the subdivision tree.

Each voxel has a 16 by 16 bit-table to store the *x,y* pixel indices of the rays that pass through it. Each bit represents a block of pixels that occupy 1/256th of the screen area. This storage method is independent of image resolution, and only requires 32 bytes of memory per voxel. Higher resolution bit-tables can be utilized if memory usage is not a constraint. Higher resolution bit-tables provide finer granularity of the rays that will be traced at each frame, with little increase in computational overhead. The ideal resolution for the bit-tables is the resolution of the image frame buffer.

When a ray enters a voxel, the bit in the voxel's bit-table that corresponds to the ray's *x,y* index is set. The voxel's bit-table may represent disjoint areas of the screen. This occurs when an object is visible to both primary viewing rays and to secondary rays, such as shadow or reflection rays, in another part of the screen (Figure 2).

3.3 Subsequent Frames

For each subsequent frame, the object space database and the subdivision structure need updating to reflect changes that have occurred since the previous frame. The entire subdivision tree is traversed, and every voxel is marked as **unchanged**. Objects that change from the previous frame are reinserted into the voxel subdivision tree, and the voxels that they affect are marked as **changed**.

If an object is deleted from the scene, the voxels in which it lay are marked as **changed**, and any references to the object are deleted from these voxels. If an object is added to the scene, the voxels in which it now lies are marked as **changed**, and references to it are added to those voxels. If an object moved or changed shape or surface attributes, both the voxels in which it lay and the voxels to which it moved are marked as **changed**, and references are added and deleted as appropriate. When marking a leaf node voxel as **changed**, the voxels above it in the hierarchy are marked as **touched**.

As long as the number of objects that change is fewer than the number of objects that remain static, the time to resort the changed objects into the subdivision structure is less than to completely rebuild the structure. This speedup is not significant, however, as the total subdivision time of adaptive subdivision algorithms is typically on the order of a few percent of the total rendering time [Jevans 89].

3.3.1 Examine the Voxel Space

The next step is to examine the voxel space to determine which pixels need retracing. A 16 by 16 bit-table representing the frame buffer is created and every bit is initialized to zero. Starting at the top level of the subdivision tree and working down, every voxel is examined. If a leaf voxel is marked as **changed**, the frame buffer bit-table is or-ed with the voxel's bit table. After the entire voxel space has been examined, the bits that are set in the frame buffer bit-table indicate which pixel blocks must be retraced.

All records of the rays that are about to be retraced must be deleted from the voxel bit-tables in case the retraced rays do not pass through those voxels in the next frame. This is accomplished by examining the voxel space a second time and clearing all bits in the voxel bit-tables that are set in the frame buffer bit-table.

The frame can now be generated. Pixels that correspond to the bits that are set in the frame buffer bit-table are retraced. All other pixels retain their color values from the previous frame.

3.3.2 Resubdivision

If the number of objects in a voxel changes significantly from one frame to another, it may be advantageous to resubdivide it. This can be determined during the examination of the voxel space. If a voxel is marked **touched** or **changed**, and the number of objects inside it has changed significantly or gone to zero, its child voxels are recursively deleted, and it is marked as **changed** and *not subdivided*. It will be examined for resubdivision on the fly when rays are being retraced.

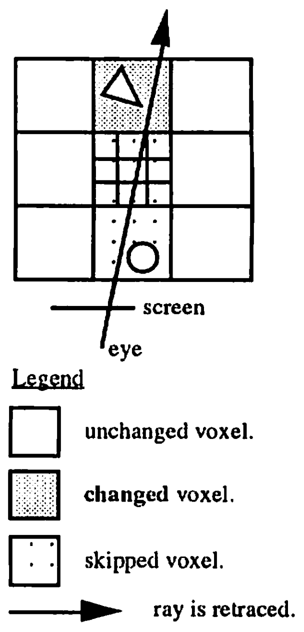


Figure 3.

3.3.3 Ray Traversal Optimization

An optimization to the traversal of viewing rays through the voxel grid can be made by storing the distance along each ray from its origin to its intersection with the visible surface. When a viewing ray is retraced, the voxel traversal algorithm can treat any non-changed voxels as empty if they are closer to the eye than this distance. Neither ray-object intersection calculations nor traversal of sub-grids need be performed for voxels that are not marked as changed (Figure 3).

4 Analysis of Animation

Since this algorithm requires that the camera remain static during an animation sequence, it is of interest to know if such sequences constitute a significant portion of computer animation. Table 1 presents statistics for the duration of time that the camera remains static for several well known animations. The timings in Table 1 are approximate however, because they do not account for cuts in the camera's point of view, nor for camera holds, which can be rendered as a single frame and replicated during filming.

For the films analyzed in Table 1, static camera sequences account for a significant portion of the animation. Naturally there are degenerate cases, such as fly-by sequences, where static camera sequences are few or nonexistent. However, for animations that include static camera sequences, a method that accelerates the ray tracing of such sequences can have a significant impact on the overall rendering time.

Animation	Running time	Static camera time	% static camera
Luxo Jr	131 sec	131 sec	100%
Red's Dream	320 sec	259 sec	81%
Tin Toy	451 sec	418 sec	93%
Pencil Test	253 sec	231 sec	91%
The Audition	309 sec	240 sec	78%

Luxo Jr, Red's Dream, Tin Toy © 1986, 1987, 1988 PIXAR.
Pencil Test, The Audition © 1988, 1990 Apple Computer, Inc.

Table 1.

5 Results

A sequence from the Apple Computer, Inc. animation "The Audition", shown at SIGGRAPH '90, was used to test the object space coherence algorithm. In this sequence a weight is dropped onto the see-saw, launching Eric the worm into the air. The motions of Eric, the see-saw, and the weight are derived from a dynamic simulation.

The sequence is 351 frames in length. The scene consists of 6000 polygons and 4 light sources. All frames were rendered at 640 by 480 resolution, with one ray per pixel, on a Silicon Graphics Personal Iris 4D/25 workstation.

The sequence was rendered one frame at a time with a ray tracer that utilizes an adaptive voxel subdivision technique. The number of pixels traced and the CPU time required to render the sequence are listed in Table 2 under the heading **Traditional Algorithm**. The sequence was then rerendered with the identical ray tracer, modified to use the object space coherence algorithm described in this paper. The CPU time, number of rays, and the ratios of these numbers compared to the traditional frame by frame approach are listed in Table 2 under the heading **Coherence Algorithm**.

Figure 4 shows several frames of the animation sequence illustrating only the pixels that were retraced by the coherence algorithm. Note that all the pixels of frame 0 are rendered by both the traditional and coherence algorithms.

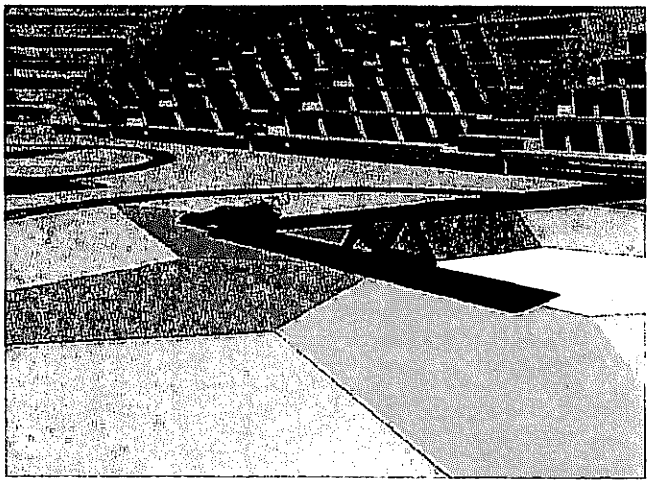
5.1 Discussion

Examining the **Entire Sequence** row in Table 2 illustrates that over the course of the animation sequence the coherence algorithm rendered only 19.35% of the rays that were traced by the traditional algorithm, and required only 26.72% of the CPU time used by the traditional algorithm, yielding a speedup of nearly a factor of four.

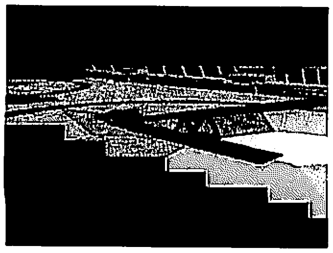
The discrepancy between the percentage of rays traced (19.35%) and percentage of CPU time (26.72%) required to render the sequence with the coherence algorithm is due to two factors. First is the overhead incurred by the coherence algorithm in building and maintaining bit-tables in each voxel and of collecting them at the beginning of each frame to determine the pixels that must be retraced. This overhead is apparent in Table 2 in the row that gives the statistics on the rendering of frame 0. The unmodified ray tracer requires 666 CPU seconds to render the frame whereas the frame coherence algorithm increases the rendering time

	Traditional Algorithm		Coherence Algorithm			
Frame #	# Rays	CPU Time	# Rays	Ratio to Traditional	CPU Time	Ratio to Traditional
0	307,200	666 sec.	307,200	1.0	780 sec.	1.171
1	307,200	669 sec.	122,400	0.3984	418 sec.	0.6248
75	307,200	684 sec.	146,400	0.4765	528 sec.	0.7719
150	307,200	649 sec.	58,800	0.1914	209 sec.	0.3220
200	307,200	634 sec.	9,600	0.0312	44 sec.	0.0694
350	307,200	655 sec.	8,400	0.0273	43 sec.	0.0656
Entire Sequence	107,827,200	63.32 hrs.	20,866,800	0.1935	16.92 hrs.	0.2672

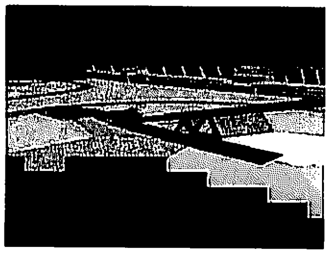
Table 2



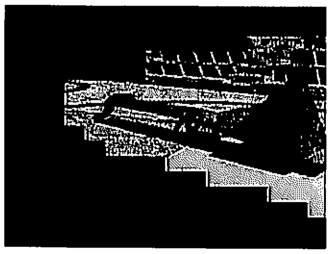
Frame 0



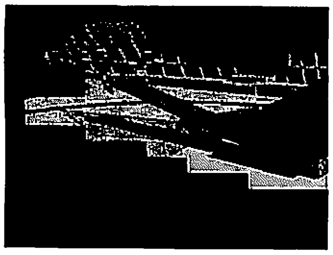
Frame 1



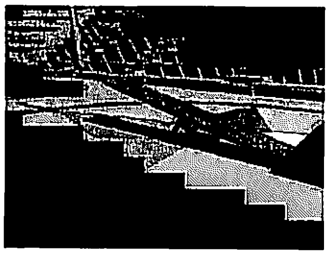
Frame 15



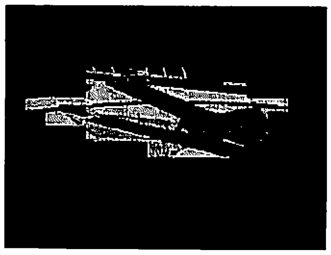
Frame 35



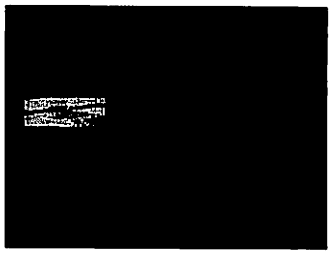
Frame 40



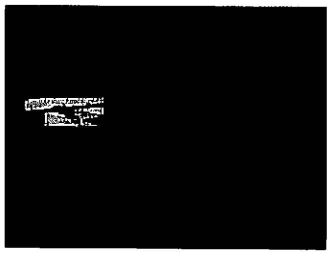
Frame 75



Frame 150



Frame 200



Frame 350

Figure 4

to 780 CPU seconds, a ratio of 1.17. This overhead is more than offset by the savings in subsequent frames.

The second source of discrepancy is due to the fact that rays are not uniform in their rendering cost. In this animation sequence, a complex object, the worm, is being retraced at each frame. The area around this complex object is more densely subdivided than the rest of the scene, requiring more traversal time per ray. The worm also has a more complex illumination model than the background model. The rays that are not retraced at each frame are typically those that intersect the background of the scene. These rays travel largely through empty voxels and intersect more simple objects, such as the tent model in this animation sequence.

6 Future Work

6.1 Inactive Voxel Collection

Adaptive spatial subdivision algorithms can reduce the amount of memory they require by taking advantage of ray coherence. When rendering an image, parts of the subdivision structure can be deleted if rays are no longer passing through them. This is common when rendering scanlines from top to bottom, as rays originating from scanlines near the bottom of the screen rarely pass through the same voxels as rays from higher scanlines. Voxels that are no longer active can be identified periodically during the rendering, and can be collected. This entails deleting the voxel's grid structure, and marking the voxel as *not subdivided*. If the assumption proves incorrect, and a ray passes through the voxel at a later time, the voxel will be resubdivided.

This idea can be extended to the temporal coherence algorithm by collecting areas of space that remain unchanged and untraversed for a number of frames. If an object inside a collected voxel changes, or a ray traverses the voxel, then it will be resubdivided.

6.2 Light Sources

When animating a light source, all rays that pass through the voxels in which it lies must be retraced. Since most ray tracers treat light sources as invisible if viewed directly, it is desirable to avoid retracing viewing rays that pass through a voxel in which a light source has moved (Figure 5). If a separate pixel index bit table for shadow rays is maintained in each voxel, then only ray trees that are affected by a moved light source are retraced.

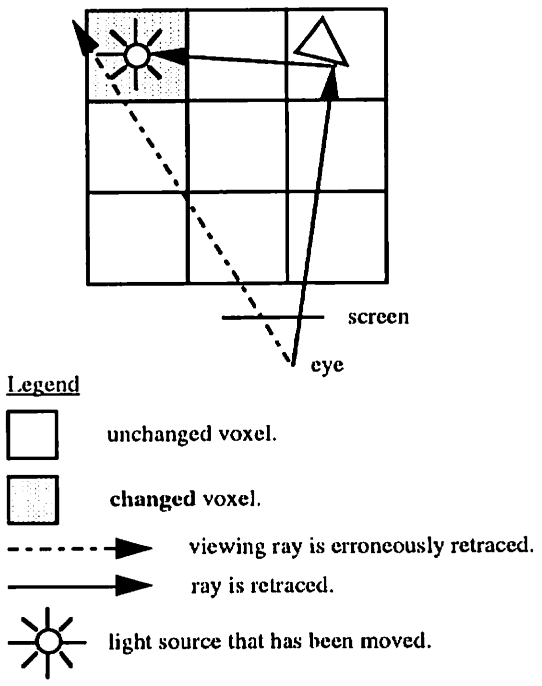

screen
eye
Legend
unchanged voxel.
changed voxel.
viewing ray is erroneously retraced.
ray is retraced.
light source that has been moved.

Figure 5.

6.3 Moving Camera

The algorithm presented in this paper may be extensible to sequences where the camera is moving, through the use of the reprojection technique proposed by Badt [Badt 88]. The 3D intersection points of the directly visible surfaces are projected onto the screen when the camera moves. If the camera moves only slightly, then the samples will not change in density, and a new image can be reconstructed from them. If the density of the samples changes, then the pixels will have to be retraced to avoid erroneous hidden surface results.

6.4 Backwards Ray Tracing

The object space coherence algorithm is useful, even with a moving camera, to accelerate backwards ray tracing techniques. Heckbert uses bidirectional ray tracing to calculate global illumination [Heckbert 90], and Watt uses backwards beam tracing to calculate light-water interaction [Watt 90]. Both methods could reduce the number of view independent rays required to render an animation sequence with non-moving light sources.

7 Conclusion

An algorithm has been presented for making use of object space temporal coherence when ray tracing animation sequences where the camera remains static. Only rays that pass through voxels in which objects have changed are traced at each frame. Memory use is independent of image resolution, and the algorithm is easily adapted to any spatial subdivision scheme.

8 Acknowledgements

Thanks to Gavin Miller for his help with the preparation of this manuscript and for the worm animation, George Drettakis and Michael Kass for their editorial help, Doug Turner for the ray tracer, and Steve Rubin for the circus tent. Many thanks to the entire Advanced Technology Group at Apple, without whom this work would not have been possible.

9 References

[Amanatides 84] J. Amanatides, "Ray tracing with cones," *Computer Graphics*, vol. 18, no. 3, pp. 129-135, July 1984.

[Arvo 87] J. Arvo and D. Kirk, "Fast ray tracing by ray classification", *Computer Graphics*, vol. 21, no. 4, pp. 55-64, July 1987.

[Badt 88] Sig Badt Jr., "Two algorithms for taking advantage of temporal coherence in ray tracing," *The Visual Computer*, no. 4, pp. 123-132, 1988.

[Carpenter 84] Loren Carpenter, "The A-buffer, an antialiased hidden surface method," *Computer Graphics*, vol. 18, no. 3, pp. 103-108, July 1984.

[Chapman 90] J. Chapman, T. W. Calvert, and J. Dill, "Exploiting temporal coherence in ray tracing," *Proceedings of Graphics Interface '90*, pp. 196-204, 1990.

[Chapman 91] J. Chapman, T. W. Calvert, and J. Dill, "Spatio-temporal coherence in ray tracing," *Proceedings of Graphics Interface '91*, pp. 101-108, June 1991.

[Chmilar 89] M. Chmilar and B. Wyvill, "A software architecture for integrated modelling and animation," *New Advances in Computer Graphics (Proceedings of Computer Graphics International '89)*, pp. 257-276, June 1989.

[Fujimoto 86] A. Fujimoto, "ARTS: accelerated ray tracing system," *IEEE CG&A*, vol. 6, no. 4, pp. 16-26, April 1986.

[Glassner 84] A. S. Glassner, "Space subdivision for fast ray tracing,"*IEEE CG&A*, vol. 4, no. 10, pp. 15-22, Oct. 1984.

[Glassner 88] A. Glassner, "Spacetime ray tracing for animation," *IEEE CG&A*, vol. 8, no. 2, pp. 60-70, March 1988.

[Heckbert 84] P. S. Heckbert and P. Hanrahan, "Beam tracing polygonal objects," *Computer Graphics*, vol. 18, no. 3, pp. 119-128, July 1984.

[Heckbert 90] P. S. Heckbert, "Adaptive radiosity textures for bidirectional ray tracing," *Computer Graphics*, vol. 24, no. 4, pp. 145-154, August 1990.

[Hubschman 82] H. Hubschman and S. W. Zucker, "Frame to frame coherence and the hidden surface computation: constraints for a convex world," *ACM TOG*, vol. 1, no. 2, pp. 129-162, April 1982.

[Jevans 89] D. Jevans and B. Wyvill, "Adaptive voxel subdivision for ray tracing," *Proceedings of Graphics Interface '89*, pp. 164-172, June 1989.

[Jevans 91] D. Jevans, *Adaptive Voxel Subdivision for Ray Tracing*, Master's Thesis, University of Calgary, 1991.

[Joy 86] K. I. Joy and M. N. Bhetanabhotla, "Ray tracing parametric surface patches utilizing numerical techniques and ray coherence," *Computer Graphics*, vol. 20, no. 4, pp. 279-285, Aug. 1986.

[Kay 86] T. L. Kay and J. T. Kajiya, "Ray tracing complex surfaces," *Computer Graphics*, vol. 20, no. 4, pp. 269-278, Aug. 1986.

[Murakami 90] K. Murakami and K. Hirota, "Incremental Ray Tracing," *Eurographics Workshop on Photosimulation, Realism, and Physics in Computer Graphics*, pp. 15-29, June 1990.

[Rubin 80] S. M. Rubin and T. Whitted, "A 3-dimensional representation for fast rendering of complex scenes," *Computer Graphics*, vol. 14, no. 3, pp. 110-116, July 1980.

[Séquin 89] C. H. Séquin and E. K. Smyr, "Parameterized ray tracing," *Computer Graphics*, vol. 23, no. 3, pp. 307-314, July 1989.

[Shinya 87] M. Shinya, T. Takahashi, and S. Naito, "Principles and applications of pencil tracing," *Computer Graphics*, vol. 21, no. 4, pp. 45-54, July 1987.

[Watt 90] M. Watt, "Light-water interaction using backward beam tracing," *Computer Graphics*, vol. 24, no. 4, pp. 377-385, August 1990.

[Whitted 80] T. Whitted, "An improved illumination model for shaded display," *CACM*, vol. 23, no. 6, pp. 343-349, June 1980.

Ray Tracing Polygons using Spatial Subdivision

Andrew Woo

Style! Division, Alias Research Inc.
110 Richmond Street East
Toronto, Ontario
M5C 1P1

1. Abstract

Ray tracing consumes a lot of computational resources to render images. This expense usually lies in the ray-surface intersection tests. If the surfaces were polygonal, then we should be able to apply more polygon-specific optimizations to partially cull intersections. Our ray tracer uses a non-memory intensive, voxel traversal intersection culler to assist in such optimizations.

Keywords: intersection culling, polygon, ray tracing, subdivision, voxel traversal.

2. Introduction

Ray tracing [Appe68] [Gold71] is widely acknowledged as a rendering approach that can produce very realistic and beautiful images [Whit80]. It is also widely known that ray tracing is very expensive computationally. Many intersection culling algorithms have been proposed to reduce this expense. However, such intersection culling algorithms do not take into consideration the nature of the primitives which they are culling.

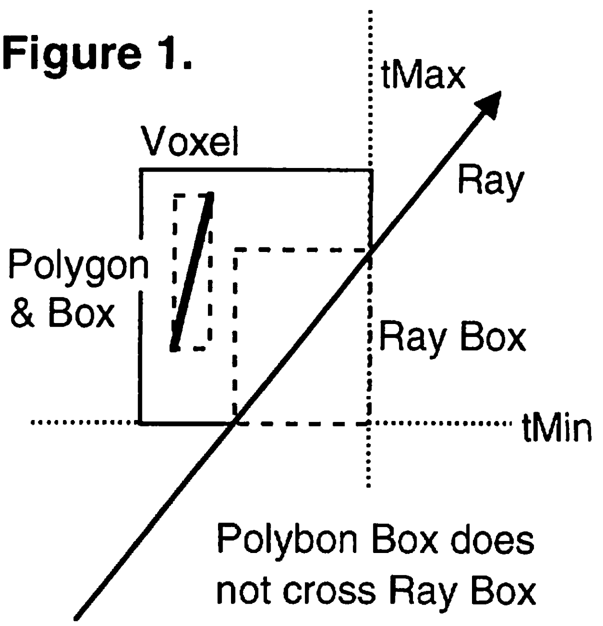
The polygon is one of the most used primitives in rendering surfaces - either as a result of tessellation of complex surfaces, or as descriptions of truncated planar surfaces. In this paper, we examine the polygon very carefully, in hopes of optimizing and reducing the need for ray-polygon intersections, while keeping the memory requirements down to a minimum. The intersection culling algorithm used to assist in such optimizations is uniform voxel traversal [Fuji86]. Our traversal implementation is taken from [Aman87] [Clea88] with ray bounding box checks [Snyd87].

3. Voxel Traveral and Ray Bounding Boxes

A popular intersection culling algorithm is voxel traversal [Glas84] [Fuji86]. Space encompassing all polygons in the scene is divided up into small 3-dimensional boxes, commonly known as *voxels*. Each voxel contains a pointer to polygons that reside in the space occupied by that voxel. Each ray generated traverses the voxel structure in-order and tests for intersection only with polygons residing in voxels that the ray pierces. Thus we hope a small candidate subset of polygons needs to be tested for intersection.

To further reduce the number of polygons that needs to be tested for intersection, Snyder and Barr [Snyd87] proposed the ray bounding box. A ray bounding box in a voxel is created from the ray segment that resides inside that voxel, bounded by the *tMin* and *tMax* extents/distances through the voxel. For each polygon in the voxel, if its bounding box does not cross the ray bounding box, then no intersection test with that polygon is necessary; see figure 1. If they do cross, then the ray-polygon intersection test is needed. This box-crossing test requires at most 6 floating point comparisons.

Figure 1.



This ray bounding box optimization has proven to accelerate the ray tracing culling process by a great deal, especially for densely populated regions distributed in a non-uniform spatial manner, in which the raw uniform voxel traversal scheme does quite miserably. See table 1 benchmarks for this evidence. However, by using ray bounding boxes, we are restricted to floating point voxel traversal schemes [Glas84] [Aman87] [Snyd87] because the *tMin* and *tMax* values need to be computed, and thus integer-only versions [Fuji86] [Clea88] cannot be used (their increase in computational speed is neglible compared to the advantages of the ray bounding box anyway).

4. Usual Ray-Polygon Intersection Process

The usual ray-polygon intersection test involves the following steps: (1) intersection against the plane on which the polygon lies to compute the hit distance t; (2) check that t is in front of the ray origin ($t > 0$) and t is not in front of any already intersected hits ($t < tHit$) - if either is false, then do not proceed any further as we have already decided that this polygon cannot be the closest visible polygon; (3) use the t value to compute the intersection point; (4) check that the intersection point lies inside the polygon: this is known as the inside-outside check.

Of all the above steps, the inside-outside check (4) is usually the most expensive. So we try to avoid this step as much as possible. One previous attempt to avoid (4) was illustrated in [Woo90], where after step (3), the intersection point is checked against the bounding box of the polygon. If the intersection point lies outside the box, then this polygon cannot possibly be hit by the ray - this check requires 6 floating point comparisons. Furthermore, there is no need to compute all the x,y,z intersection points before checking with the bounding box. Computing the x intersection point followed by checking with the x extents of the bounding box, then repeat with the y and z extents, will be all that much more efficient. This optimization appears to be very effective for tessellated polygons.

5. Order of Candidates for Intersection

In step (2), we also make sure that the t value of the current polygon is not in front of any intersection hits *tHit* that have already taken place, i.e., $t < tHit$. If $t > tHit$, then even if this ray does intersect the polygon, it will not be the closest visible polygon. So why bother with steps (3) and (4)? This leads us to think that it is advantageous to have the closest visible polygon tested for intersection near the beginning and all other candidate polygons can be trivially dismissed from the $t < tHit$ check (as well as the advantage to be described in section 6).

5.1. Dynamic Updating of the Database

For each voxel, there exists a linked list of candidate polygons that occupy the voxel. Ray-polygon intersection tests occur in-order through the linked list. For our optimization, when a ray intersects the closest visible polygon inside the voxel, that polygon is shifted up to the beginning of the linked list. Future rays that pierce the voxel may have the same visible polygon but now have the advantage of intersecting the visible polygon first (or close to first). Then many other ray-polygon intersection tests are rejected at step (2).

This optimization should theoretically be quite effective for shadow determination. It is common practice to assume that what the previous shadow ray hits may be true for the current shadow ray [Hain86]. In addition, shadow rays only need to determine if there exists an intersection hit or not. Thus updating the voxel linked lists might lead to an earlier intersection hit for future neighbouring shadow rays. In addition, this is better than just keeping one polygon pointer to what was previously intersected [Hain86], since neighbouring shadow rays might intersect polygons A, then B, then A, then B, etc. in that order. With our optimization, we should detect intersection within the first few tests of that voxel. In addition, this optimization would be a nice complement to another shadow culler [Pear91] in which triangles residing in the last shadow ray hit voxel are intersected first.

However, we found that the linked list updates for shadow rays do not perform that well. Perhaps the [Hain86] [Pear91] optimizations have already done quite a lot of the work in our implementation already. And having many lights, such linked list updates may prove to be quite useless. Thus, this linked list update is only done for non-shadow rays.

5.2. Almost Hit Cases

In the previous subsection, we only shifted the hit cases up to the front of the voxel's linked list so that future rays will hit the visible polygon near the beginning. However, chances for the same visible polygon are not as likely due to the small tessellated polygons. Thus, we should shift pointers up to the front of the linked list for nearly hit polygons as well. Then we will be able to get even better results on reaching future visible polygons faster.

An almost hit case will be one that returns a *no hit* intersection while in step (4) of the ray-polygon process (inside-outside check), and after the [Woo90] optimization check. This optimization check is quite reliable because it eliminates many candidate polygons, except ones that the ray is close to.

5.3. Using the RayID Effectively

In the previous subsections, we needed to do some linked list shifting. The *rayID* [Aman87] (first proposed to eliminate multiple intersections with the same object in a voxel traversal environment) can be used to select better candidates for intersection without such shifting. For a linked list of candidate polygons to intersect, a 2-pass walk through of the list is needed. The first time the linked list is traversed, only the polygons whose *rayID* is quite close to the current ray's *rayID* are intersected. Then the second time the list is traversed, the remainder of the candidate polygons are intersected.

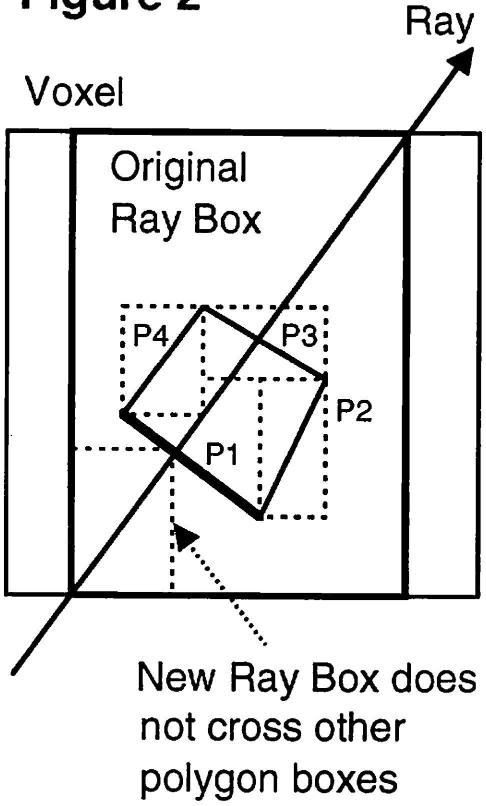
We do this optimization because the closer the polygon *rayID* is to the current ray's *rayID*, the indication that previous rays have attempted to intersect with this polygon. Thus it is more probable that this polygon may be hit by the current ray. A polygon's *rayID* that is very different from the ray's *rayID* indicates that the polygon has not been intersected lately by previous rays - thus it is likely that the polygon will not be the visible polygon.

On trying out this optimization, it appeared that the *rayID* is not really a good indicator of better candidates, especially when in section 6, dynamic ray boxes are used. Thus, this optimization was removed from our implementation.

6. Dynamic Ray Bounding Box

For each voxel, Snyder and Barr [Snyd87] suggested a box-crossing test between the ray bounding box and each polygon's bounding box. If they do not cross, then we know that intersection with the polygon must fail without any actual ray-polygon intersection tests. We can do a little better: once we get any intersection *tHit* (not just the closest one) with the ray, we can also dynamically reduce the size of the ray bounding box. In other words, the ray bounding box is bounded by [*tMin*, *min*(*tHit*,*tMax*)], instead of [*tMin*, *tMax*] †. With the section 5 optimizations, we hope that the closest visible polygon with hit *tHit* will be encountered near the beginning of the voxel's linked list. As a result of this, the ray bounding box can be adjusted earlier and more ray-polygon intersection tests can be avoided from the box-crossing test.

Figure 2



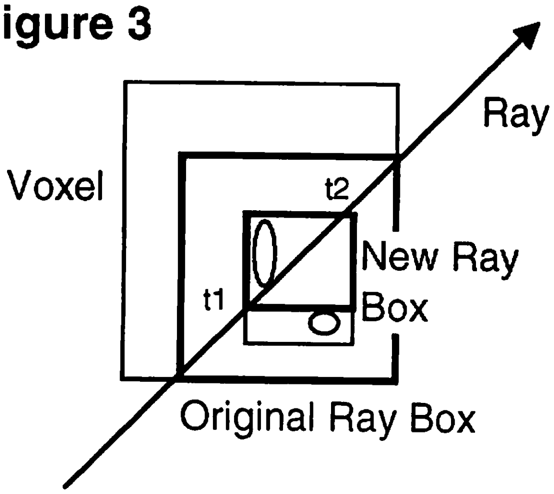
† If *tHit* > *tMax*, then the hit intersection point resides in a subsequent voxel. That voxel, when reached, can be bounded by [*tMin*, *tHit*].

This new ray bounding box does not require additional computation, since it is already done in the ray-polygon intersection hit. In addition, note that this optimzation does not delay intersection, but is a guaranteed culling step; see next subsection's pseudo code. It can be interpreted as a form of object coherence, where we base our assumptions of visibility on a previous neighbouring ray. For example, in figure 2, if we intersect the visible polygon P1 first as a result of the section 5 optimizations, then all the other polygons P2, P3, P4, (which may belong to the same convex surface) residing in the same voxel can be trivially rejected for intersection due to the newly adjusted ray box. If the ray bounding box is not adjusted, then intersections with all those polygons would be needed.

Note that the starting ray bounding box for a voxel does not need to be bounded by [*tMin*, *tMax*] either. For each occupied voxel traversed, we added a ray-box intersection check with the bounding box containing all the polygons (bounded by the voxel) inside the current voxel - refer to this bounding box as *B*. This is done in hopes that the polygons only occupy a small region of the space inside the voxel, and that the ray might miss all of them - we only do this intersection check if *B* is smaller than the voxel itself. If the ray does not intersect *B*, then we just continue traversal. However, if the ray does intersect *B*, we have to perform box crossing and possibly intersection tests with those polygons inside the voxel. Then if the ray intersects *B* at distances t_1, t_2, we can start off with a smaller ray bounding box bounded by [t_1, t_2], where $tMin \le t_1 < t_2 \le tMax$. Note, in figure 3, with this new ray box formed by t_1 and t_2, we do not have to intersect against the bottom sphere.

In the case of second generation rays, it is usually that t_1, $tMin < 0$. Then we bound the ray bounding box by [0, t_2]. This usually avoids the intersected surface, if convex, to test for reflection intersections against itself. Self-mirror reflections cannot occur for convex surfaces.

Figure 3



6.1. Some Pseudo Code

/* Standard ray bounding box scheme for each voxel */

```
rayBox.bound1 = O + voxel->tMin * D;
rayBox.bound2 = O + voxel->tMax * D;
for (each polygon in voxel)
    {
    if (boxesCross (rayBox, polygonBox))
        hit = intersectPoly (&tHit,
            &intersectPoint, polygon);
    }
```

/* New dynamic ray bounding box scheme */

```
rayBox.bound1 = O + max[0,t1] * D;
rayBox.bound2 = O + t2 * D;
for (each polygon in voxel)
    {
    if (boxesCross (rayBox, polygonBox))
        hit = intersectPoly (&tHit,
            &intersectPoint, polygon);

    /* section 5 optimization */
    if (hit || almostHit)
        place polygon at the front of
        the voxel list;

    /* section 6 optimization */
    if (hit && tHit < t2)
        rayBox.bound2 = intersectPoint;
    }
```

6.2. Avoiding Ray Bounding Box Evaluations

In the above pseudo code, the *boxesCross* routine takes at most 6 floating point comparisons. However, this is needed for each polygon in a voxel. In some instances, these ray bounding box evaluations are known ahead of time to not help speed up the ray tracing process, then we might as well do the ray-polygon intersection immediately (without any *boxesCross* tests). This is when the ray crosses a voxel that results in a ray bounding box close to the size of the voxel. We can detect this situation by evaluating $t_2 - t_1 > tMaxVoxel$ for each voxel. If true, then we do not bother with the ray bounding box checks, where $tMaxVoxel = \beta \sqrt{X^2 + Y^2 + Z^2}$ (which is a constant), X,Y,Z represent the dimensions of each voxel, and β is some value close to and less than 1 (a good choice for β is 0.75). Note that if β is 1, then *tMaxVoxel* is just the maximum ray segment that can pass through the voxel.

Then we avoid the box crossing test for this voxel unless the ray bounding box can be reduced by the section 5 and 6 optimizations.

6.3. Multiple Ray Bounding Boxes

If there are a large number of polygons in the voxel, an obvious optimization that can be done here is to have multiple ray bounding boxes for this voxel. Then we have a step-case of smaller ray bounding boxes. If there are r ray bounding boxes, then the voxel's linked list of polygons need to be traversed r times, each time their bounding boxes checked against the current ray bounding box.

This was implemented and tested on our raytracer, but found the results to be quite disappointing and thus was not included in the final raytracer code. It appeared that with multiple ray bounding boxes, less ray-polygon intersections took place. However, many ray-polygon intersections were just delayed by the many box crossing tests. Then the box crossing tests dominated the processing time.

7. The Ray-Plane Intersection

The ray-plane intersection (step 1) needs to be optimized to improve the overall ray-polygon intersection test. The usual computation of t with a plane (or polygon) is:

$$t = \frac{d - N \cdot O}{N \cdot D}$$

where the plane equation is defined by $N \cdot P = d$, P is a $<x,y,z>$ variable triplet, N is the surface normal, and the ray (thus the ray-plane intersection point in step 3) is defined by $O + tD$. We will show that the t evaluations should take 6-8 floating point operations under some circumstances.

7.1. For Second Generation Rays

With second generation rays being shot, we can reuse some previous results to save computation. In other words, for a reflection ray, some of the ray-plane computation can be reused from its parent/cast ray.

We will subscript all cast ray information with c and reflected ray information (extendible to refraction and shadow rays as well) with r. We can expand $d - N \cdot O_r$ to $d - N \cdot (O_c + t_c D_c)$, where t_c is the t value for the closest visible polygon in the cast ray. Then we get $d - N \cdot O_c - t_c N \cdot D_c$. As a result, only an additional multiplication and subtraction are needed to compute the numerator of t_r in step (1): we already know the values of t_c, $d_1 = d - N \cdot O_c$ and $d_2 = N \cdot D_c$ from the cast ray. The reflection ray numerator is then simply $d_1 - t_c d_2$.

After some implementation and testing, we found that this optimization is not worth the trouble due to the small number of same cast and reflection ray hits, and due to the complicated information management within our scheme.

7.2. For Cast (First Generation) Rays

For first generation or cast rays, we notice that the numerator of the t evaluation is always a constant for the same triangle (assuming a perspective view). In other words, $d - N \cdot O_c$ is the same througout each triangle. However, it is far too memory consuming to store the numerator for each triangle in order to save dot product evaluations.

We can preprocess and translate the entire database so that the eye origin resides at (0,0,0). Then $N \cdot O_c = 0$ and the numerator of the t evaluation is d for all triangles, without the need for any storage. Thus $t = d / N \cdot D_c$.

There are other advantages to this translation as well. The ray-plane intersection point computation is simplified from $O_c + tD_c$ to just tD_c. We can use this tD_c to simplify ray bounding box computations for $tMin*D_c$ and $tMax*D_c$ as well. However, we cannot apply this optimization when depth of field effects (along the flavour of distribution ray tracing [Cook84]) are to be generated.

7.3. Dynamic Clipping of t Values

In step (2) when the t value is computed, we need to check that t is in front of the ray origin. In most implementations, a constant fudge value is used to evaluate this: e.g., $t > 0.0001$. However, we can use dynamic clipping of the t extents. We should check that $t > tMin$ instead - actually, it is even better to check that $t > t_1$ (since $t_1 > tMin$).

This optimization allows us to omit steps (3) and (4) if $t < t_1$. This is because the intersection point is far beyond the polygon (beyond the voxel, in fact) that we know it cannot possibly be an intersection hit. With more tight values like t_1 to clip against, the better the chance that steps (3) and (4) can be omitted.

8. Implications of Voxel Subdivision

One open and difficult question in uniform voxel traversal approaches is the subdivision level necessary to get an optimal overall ray tracing performance. If the subdivision level is too deep, then we pay for more traversal and extensive memory costs, but gain the advantage of only needing to deal with small number of surfaces in each voxel. If the subdivision level is not deep enough, we pay for the cost of having to perform more ray-surface intersection tests - this can significantly slow down the ray tracing process. Devillers [Devi88] attempted to answer this question with an analytic solution R for uniformly sized voxels. Subramanian and Fussell [Subr91] also attempted to answer this question in a similar fashion. However, both papers always assumed a $R{\times}R{\times}R$ subdivision scheme, instead of a general $X{\times}Y{\times}Z$ scheme. In addition, there are so many variables (though spatial distribution and number of polygons play a major role in this analysis) which we must consider that it cannot all be encapsulated in an analytic equation. And is it worth the effort to compute this complicated and expensive solution anyway?

8.1. Previous Benchmarks

With ray bounding boxes [Snyd87], it seems that we are less reliant on the voxel subdivision as compared to the raw voxel traversal approach; the ray bounding boxes act as second level cullers in case many surfaces need to be tested for intersection. And since voxels do take up a lot of memory, it seems that we should consider $X{\times}Y{\times}Z$ subdivision schemes that are small. The justification for minimal subdivision can be seen in table 1, where ray bounding boxes (without our optimizations discussed here) were used to accelerate ray tracing on a SUN 3/280 with fpa, and 8 megabytes of memory. The benchmarks were done on a University of Toronto ray tracing program named *optik* with true spheres (not tessellated into many polygons), benchmarked in 1988. The image used was taken from the Haines' sphere flakes image [Hain87], where a densely populated environment distributed in a non-uniform manner is created due to a large floor and a small, concentrated set of spheres. Note that *Ray Box* indicates the CPU minutes taken to ray trace with the ray bounding box and uniform voxel traversal; *Raw Traversal* indicates the CPU minutes taken to ray trace with only uniform voxel traversal.

#Sphere	Grid Res	Image Res	Ray Box	Raw Traversal
7382	40×40×40	512×512	163	391
7382	50×50×50	512×512	142	270
7382	60×60×60	512×512	138	220
7382	70×70×70	512×512	133	199

Table 1: Ray Box vs. Voxel Traversal

8.2. An Approximate Subdivision Level

Another reason we sought a memory conservative voxel subdivision is due to our main platform - the Mac II. Memory conservation is so essential, we can only assume a maximum configuration of 8 megabytes of memory on the machines. This is why many of our optimizations do not take up additional memory and help reduce the usage of memory as well: an alternative to faster but more memory intensive, voxel-based cullers [Jeva89].

We do not try to look for an optimal subdivision level; an approximate one keeping the subdivision minimal should do. Our subdivision scheme to be described below works well in general for polygons. We consider only one main variable: the total number of polygons in the scene - label this n. Let $m = n^{1/3}$, assuming a rather uniform distribution of polygons throughout 3-space. Then we compute the bounding box surrounding all polygons in the scene. Let the spans of the bounding box extents (difference between the maximum and minimum extents) be labelled s_x, s_y and s_z for each of the axes. Then, taking into consideration the spans that occupy the voxel space,

$$X = \frac{s_x}{maxS}\,m,\quad Y = \frac{s_y}{maxS}\,m,\quad Z = \frac{s_z}{maxS}\,m.$$

where $maxS = max(s_x, s_y, s_z)$. This linearity provides us with more cubical voxels than the standard $R{\times}R{\times}R$ subdivision. More cubical voxels give a good distribution of polygons within voxels without using up the excess memory imposed by a $R{\times}R{\times}R$ subdivision scheme.

8.3. Order Complexities of the Subdivision Level

With this voxel subdivision strategy, we have placed an upper bound on the memory consumption. At worst, the number of voxels is $m{\times}m{\times}m$, which actually equals n. And at worst, each voxel will contain pointers to n polygons. Thus the upper bound memory usage is $O(n^2)$ (actually, we can lower this upper bound to $O(n^{5/3})$ for planar polygons if a smart insertion into voxels is done). However, having each voxel containing n polygons is very unrealistic: this means that all the modelled polygons occupy a large chunk of 3d space. The best case memory usage is O(n), indicating that a polygon resides totally within a fixed number of voxels, which may very well be the norm for tessellated polygons.

Since the complexity of the ray bounding boxes is clearly O(n), then the above paragraph's complexities also hold true for the entire ray tracing intersection culler memory requirements.

9. Testing and Analysis

The testing was done on a standard 68030, 40 MHz, *Mac IIfx* running under the MPW3.2 environment on MultiFinder 6.0.5 with 8 megabytes of memory. The C code was compiled under the standard MPW C compiler. In our implementation, all surfaces are tessellated into triangles and efficiently ray traced in barycentric coordinates. Our raytracer is a ported and re-coded version of Alias' raytracer product, version 3.0. This ported version serves as the raytracer for the Sketch! product on the Mac.

Many previous papers on intersection culling algorithms appeared only interested in the number of ray-surface intersections that were done. Their only aim was to lower the number of such intersections. However, the computation to avoid such intersections may become even more costly than the actual intersection itself. Intersection culling research is at the point now where this number alone has become a less important indicator for speeding up the ray tracing process. Thus, in the upcoming testing sections, the number of intersections done are not stressed in our benchmarks.

A worst-case breakdown of our ray-triangle intersection scheme is as follows: step (1) requires 13 flops, step (2) requires 2 flops, step (3) requires 6 flops, the extra bounding box check [Woo90] requires 6 flops, and step (4) done in barycentric coordinates requires 25 flops. This comes to a total of 52 floating point evaluations per triangle. With the section 7 optimizations, the floating point count for cast ray-intersections is lowered to 43.

9.1. General Testing

The benchmarks in table 2 are listed in total CPU minutes and seconds to render the 640×480 images, and exactly 1 sample per pixel is taken. On average, it appears that we get about 9-14% improvement with our optimizations over the old timings (*New Time* over *Old Time*), where the old timings represent the basic Snyder and Barr culler implementation [Snyd87] and using the subdivision level calculated via section 8's method. Considering how much superior ray bounding boxes are over uniform voxel traversal (as can be seen in table 1), the 9-14% improvement is not too bad. Also note that *%Intersect* is the percentage of ray-polygon intersections saved with our optimizations.

Everything in the scene is made mirror reflective, with a maximum recursive reflection depth of 3. Note also that the lamp image is non-uniformly/sparsely distributed due to the large floor on which the lamp lies on. The lamp image is densely populated due to the small nuts and bolts, as well as the duplicated lamp bowl on the inside and outside.

Image	#Tri	Grid Res	%Intersect	Old Time	New Time
Spheres	3500	15×9×5	13.6%	7:55	6:43
Room	5182	17×17×16	10.2%	37:20	34:01
Lamp	29062	30×10×30	14.9%	25:44	23:27

Table 2: General Optimization Benchmarks

9.2. Object Coherence Testing

We suspect that with a higher sampling rate will come better improvement results for our optimizations. This is mainly due to the assumption that object coherence optimizations in sections 5 and 6 are more likely to get similar hits between subsequent rays (which will provide the majority of the speedups in this paper). The *lamp* image, at resolution 640×480, is being used to test out this assumption (see table 3), where a 6.7% (8.9 – 2.2) improvement jumps to 11.3% (12.2 – 0.9) with more sampling. Note that *sampling* indicates the maximum sampling rate in an adaptive sampling scheme [Whit80], *Tri Time* represents the timings for the optimizations mentioned in section 7, and *New Time* represents the timings for all our optimizations.

Sampling	Old Time	Tri Time	%Improve	New Time	%Improve
1×1	25:44	25:10	2.2%	23:27	8.9%
2×2	37:32	37:11	0.9%	34:07	9.5%
3×3	61:28	60:55	0.9%	54:02	12.2%

Table 3: Lamp Image with Levels of Anti-Aliasing

9.3. Voxel Subdivision Testing

Our voxel subdivision scheme proposed in section 8 needs to be verified, for it is difficult to accept that such minimal subdivision is sufficient in many cases. Most programmers implementing a uniform voxel traversal scheme usually employ a much deeper subdivision level. Table 4 illustrates an example for the spheres image. Note that *space usage* indicates the total memory usage by the program, and note the alarming increase in memory as subdivision level increases for even such a simple scene.

Grid Res	Time	Space Usage
15×9×5	6:43	1,003,852 bytes
15×15×15	6:44	1,221,752 bytes
20×20×20	6:50	1,331,768 bytes
30×30×30	7:12	1,801,816 bytes

Table 4: Sphere Image with different Subdivision

In addition, note the comparatively small speed difference in table 1 between the different subdivision levels. With our optimizations and having only to ray trace polygons, we expect that the difference will even be narrowed more.

10. Optimization Extensions

The list of optimizations mentioned in this paper can be trivially extended to other surface types as well as culling approaches. For example, the main surface type we considered here is the polygon. However, the use of object coherence with respect to the ray bounding box (sections 5 and 6) can be applied to other surface types such as general quadrics, parametric and implicit surfaces, etc. In fact, the speedups should be even superior due to the more expensive ray-surface intersection routines for the complex surfaces and the availability of more object coherence (as compared to tiny polygons).

The object coherent ray bounding boxes can be applied to most voxel-based cullers [Glas84] [Aman87] [Synd87] [Jeva89]. Even with hierarchical voxel structures [Glas84], we can apply these optimizations and reduce the voxel subdivision or in case the maximum depth of the voxel is reached but the voxel is still over-populated. Furthermore, if voxels are created on the fly [Jeva89] as needed, as opposed to all preprocessed voxels, then a much higher polygon count should be the limit used before further voxel subdivision is done.

11. Conclusions and Further Discussions

We have listed some simple but rewarding optimizations that can be easily achieved for ray tracing polygons. They have the advantage of requiring neither additional memory nor substantial additional floating point computation. Some of the optimizations can be applied to other surface types and intersection cullers as well.

Another optimization idea is to generate *cartoon reflections* for planar polygons. Such reflections place a decay/fading factor on its intensity, where reflections have no visible effect after a certain maximum distance *maxDist*. A simple decay factor can be $[(maxDist - tHit)/maxDist]^k$; where $tHit < maxDist$. Voxel traversal and ray-polygon intersections will be halted beyond *maxDist*, but approximate reflection information will usually already have been generated. How acceptable and useful is this approximation? Could this decay apply to shadows to fake ambience (shadow intensity as a function of the distance to the closest occluding object) as well? See cartoon reflections image, where the floor has parameters $maxDist = 5$ for reflections, $maxDist = 10$ for shadows, and $k = 1$, i.e. linear decay.

Final thought: if our subdivision level is really quite small and most polygons fit in few voxels, then do we really need the *rayID* concept in this environment? Based on other experiences [Sung91], the *rayID* may not always accelerate the ray tracing process, probably due to extensive memory usage - a 32 bit flag is attached to each polygon. A possible alternative is to make the *rayID* an unsigned short (16 bits), and reinitialize all *rayID*'s after 65535 rays are shot. This was implemented ontop of our raytracer but found no noticeable speedups. Other alternatives are needed...

12. Acknowledgements

Thanks to Andrew Pearce, Steve Chall, Maria Raso (of Alias Research), and Kelvin Sung (University of Illinois) for proofreading and providing valuable suggestions to this paper.

13. References

[Aman87] J. Amanatides, A. Woo, "A Fast Voxel Traversal Algorithm for Ray Tracing", Eurographics, August 1987, pp. 1-10.

[Appe68] A. Appel, "Some Techniques for Shading Machine Renderings of Solids", Proc. AFIPS JSCC, vol. 32, 1968, pp. 37-45.

[Clea88] J. Cleary, G. Wyvill, "Analysis of an Algorithm for Fast Ray Tracing Using Uniform Space Subdivision", Visual Computer, July 1988, pp. 65-83.

[Cook84] R. Cook, T. Porter, L. Carpenter, "Distributed Ray Tracing", Computer Graphics, 18(3), July 1984, pp. 137-145.

[Devi88] O. Devillers, "The Macro Regions: An Efficient Space Division Structure for Ray Tracing", Rapport de Recherche du Laboratoire d'Informatique de l'Ecole Normale Superieure, Paris, November 1988.

[Fuji86] A. Fujimoto, T. Tanaka, K. Iwata, "ARTS: Accelerated Ray-Tracing System", IEEE Computer Graphics and Applications, 6(4), April 1986, pp. 16-26.

[Glas84] A. Glassner, "Space Subdivision for Ray Tracing", IEEE Computer Graphics and Applications, 4(10), October 1984, pp. 15-22.

[Gold71] R. Goldstein, R. Nagel, "3-D Visual Simulation", Simulation, January 1971, pp. 25-31.

[Hain86] E. Haines, D. Greenberg, "The Light Buffer: A Shadow Testing Accelerator", IEEE Computer Graphics and Applications, 6(9), September 1986, pp. 6-16.

[Hain87] E. Haines, "A Proposal for Standard Graphics Environment", IEEE Computer Graphics and Applications, 7(5), May 1987, pp. 3-5.

[Jeva89] D. Jevans, B. Wyvill, "Adaptive Voxel Subdivision for Ray Tracing", Graphics Interface, June 1989, pp. 164-172.

[Pear91] A. Pearce, D. Jevans, "Exploiting Shadow Coherence in Ray Tracing", Graphics Interface, June 1991, pp. 109-116.

[Snyd87] J. Snyder, A. Barr, "Ray Tracing Complex Models Containing Surface Tessellations", Computer Graphics, 21(4), July 1987, pp. 119-128.

[Subr91] K. Subramanian, D. Fussell, "Automatic Termination Criteria for Ray Tracing Hierarchies", Graphics Interface, June 1991, pp. 93-100.

[Sung91] K. Sung, "A DDA Octree Traversal Algorithm for Ray Tracing", Eurographics, September 91, pp. 73-85.

[Whit80] T. Whitted, "An Improved Illumination Model for Shaded Display", Communications of the ACM, 23(6), June 1980, pp. 343-349.

[Woo90] A. Woo, "Fast Ray-Polygon Intersection", *Graphics Gems*, ed. A. Glassner, Academic Press, August 1990, pp. 394.

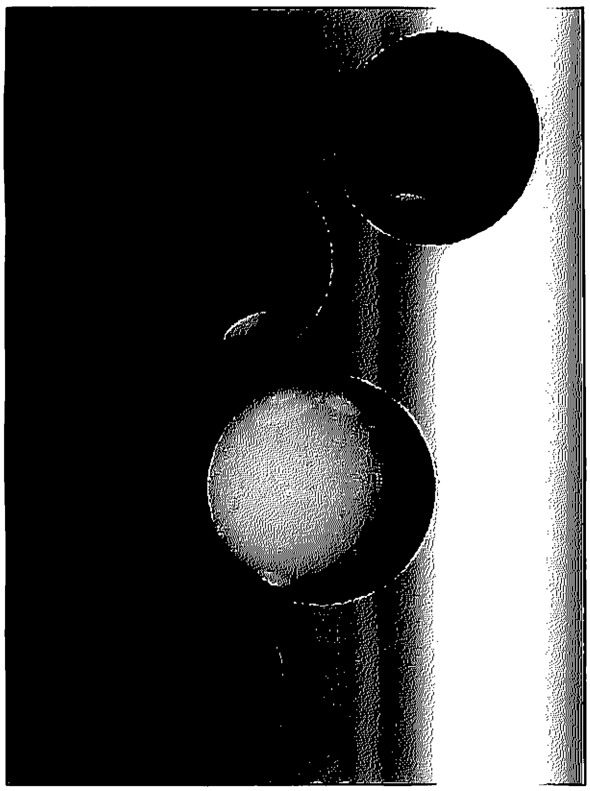


Image 1: Spheres

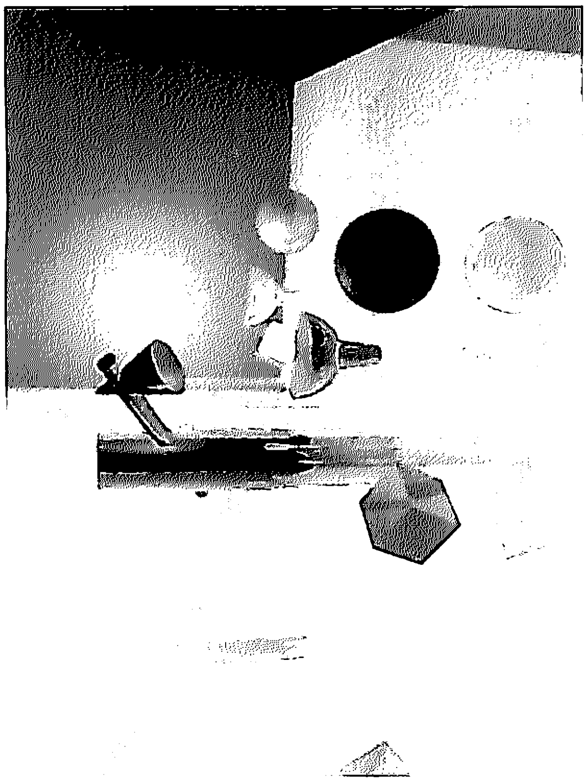


Image 2: Room

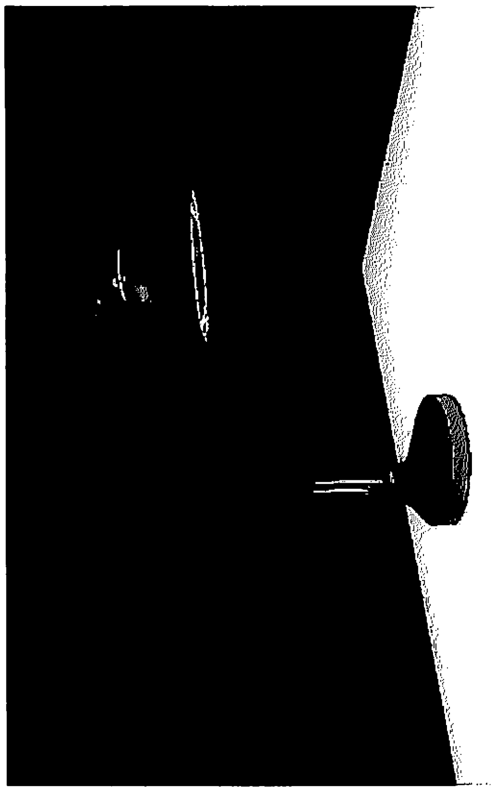


Image 3: Lamp

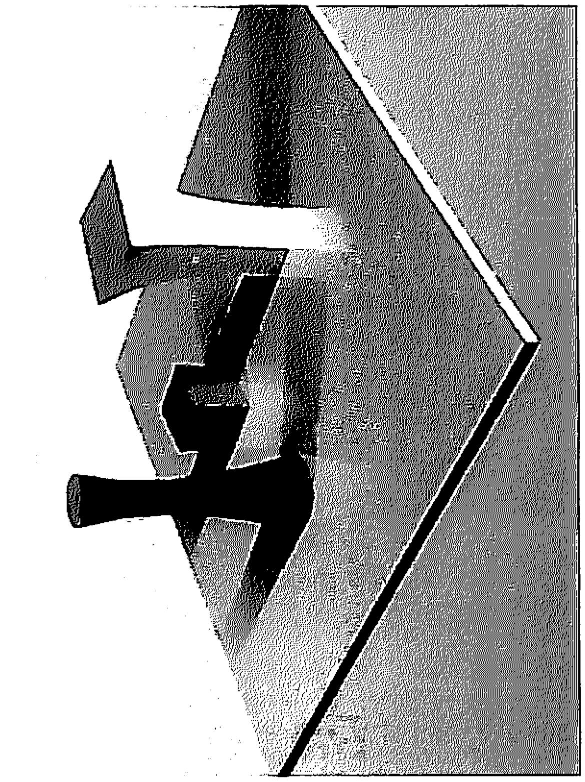


Image 4: Cartoon Reflections

Parametric Height Field Ray Tracing

David W. Paglieroni
Sidney M. Petersen
Loral Western Development Laboratories
3200 Zanker Road
San Jose, CA 95134

Abstract

Height field ray tracing is one approach to photo-realistic visualization of terrain. *Parametric methods*, which fall into a new class of height field ray trace methods, are introduced[1]. The height field is horizontally sliced into evenly spaced cross-sections. For each cross-section, the Euclidean distance from each point to the nearest point where the slice cuts through terrain is computed off-line. These planes of distance values are condensed into *parameter planes* that encode cones of empty space above each height field cell, where cone width is bounded by the terrain relief. Parametric ray tracing occurs along intersections between rays and these cones. Parametric methods are shown to be memory efficient and often much faster than the other popular height field ray trace methods.

Keywords: Height Field, Ray Tracing, Distance Transform, Parameter Planes, HDDT Profile, Incremental Methods, Hierarchical Methods, Parametric Methods

1. Introduction

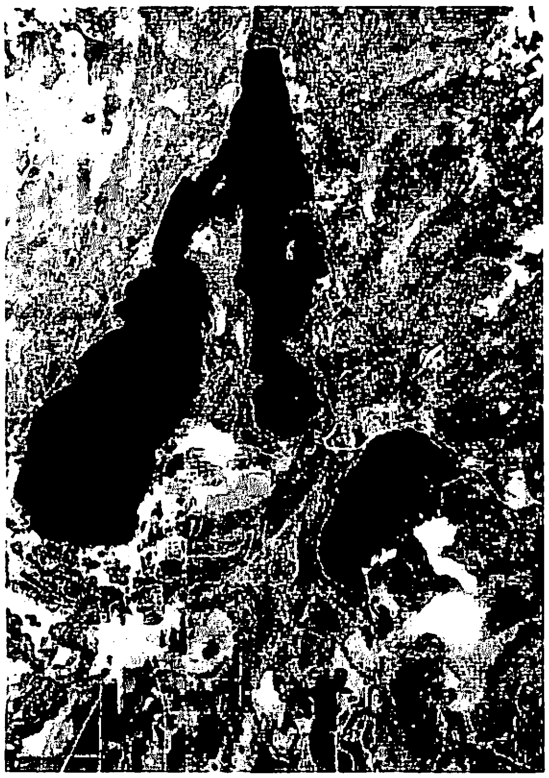
The probability of mission success associated with use of a mission planning and rehearsal system and the effectiveness of flight simulator training can be dramatically increased by introducing photo-realism. The most popular approach to photo-realistic visualization of terrain is probably the photo-textured polygon approach. This approach represents terrain as a continuous surface of planar polygonal facets. Warp coefficients that map pixels in simulated views to the source image grid are computed for each polygon so that warping can be used to generate simulated views in real-time. Occlusion is handled by some form of hidden surface removal.

Height field ray tracing is another approach to photo-realistic visualization of terrain. There are several important distinctions between the photo-textured polygon and height field ray tracing approaches. The ray tracing approach represents terrain with a simple raster data structure of height samples rather than with a vector data structure of connected polygon vertices. This allows the terrain to be modeled as a continuous surface of connected patches that can be curved to enhance realism. The ray tracing approach can be parallelized on a pixel-by-pixel basis because it does not use hidden surface removal, which is inherently sequential, to handle occlusion. However, height field ray tracing is less popular because it is widely believed to be too computationally intensive. Methods for accelerating height field ray tracing are reviewed below and an efficient new method for height field ray tracing is introduced in section 2. A source image and photo-realistic perspective view generated from it by height field ray tracing are shown in Fig.1.

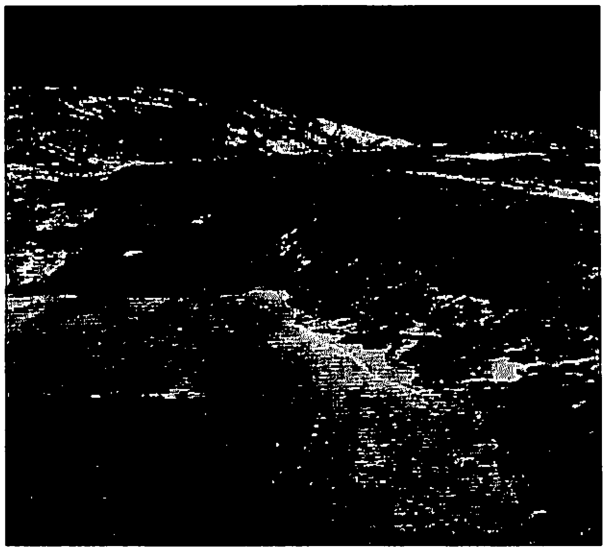
1.1 Height Field Ray Tracing

The database used for height field ray tracing contains source images, height fields and perhaps data derived from height fields, where *height fields* are arrays of evenly spaced height samples. During height field ray tracing, portions of the source image may be swapped into RAM from disk but for any local area of interest, the height field and any data derived from it should reside completely in RAM. The height field ray tracing approach maps intensities of source image pixels onto a simulation display. The math model associated with the simulation is used to characterize lines-of-sight or rays through pixels in the simulation display. If, for example, the simulation is a perspective view, then the simulation model is that of a frame camera. Height field ray tracing

[1] Research sponsored by Loral Western Development Labs. US and international patent applied for on parametric ray trace process.



(a)



(b)

Fig.1 McCall Idaho: (a) reduced resolution copy of 5200 x 7200 source image (b) 512 x 512 oblique perspective view generated from (a) by ray tracing every pixel.

determines where rays first pierce terrain surfaces modeled as interpolated (bilinearly interpolated in this study) height fields. These *pierce-points* have ground coordinates that can be analytically mapped to source image pixels by applying the math model of the source image.

Height field ray tracing is an inherently numerical process since the point where a ray first pierces an interpolated height field surface can generally be found only by tracing the ray out from the viewpoint until a pierce-point is encountered. This process has several components. First, the rays are characterized mathematically. The height field bounding box is then used to isolate the segment to be traced on each ray. Traversal progresses from point to point on the ray in ray trace steps. The height of the ray point at the end of each ray trace step is compared to the height of the height field cell that it lies above. Each cell is defined by four adjacent height field samples that lie at the corners of a rectangle. The height of the cell will be taken as the height of its tallest corner. If the ray could possibly pierce the cell that it lies above, a pierce-point calculation is performed [1]. If the ray does not pierce that cell, ray tracing continues until a pierce-point is found or the ray exits the height field bounding box.

The ray tracing approach to photo-realistic visualization of terrain can be accelerated by reducing the number of rays to be traced or by accelerating the ray trace process. The number of rays to be traced can be reduced by tracing rays along lines-of-sight through every so many pixels on the simulation display and tile warping in between. For *tile warping*, ground coordinates associated with every nth pixel in the simulation display are computed by height field ray tracing. The source image math model is used to analytically map these terrain points onto the source image display. Warp coefficients that map tile pixels to source image pixels are determined for each rectangular tile in the simulation display. Source pixels associated with each tile pixel are then computed by tile warping, which costs less than ray tracing. Acceptable tile sizes are dictated by the nature of the terrain relief within tile field of view.

To accelerate the ray trace process, *ray vertical coherence* is sometimes exploited [3-4]. Rays that lie in the same vertical plane (i.e., any plane normal to the ground plane) project to the same line on the ground and are said to be vertically coherent. In perspective views, vertically coherent rays pass through points that all lie on the same line in the focal plane and they are arranged in order of decreasing steepness along that line. Ray tracing can begin from the x (east) and y (north) associated with the pierce-point of the next steeper vertically coherent ray provided that the terrain surface is modeled as a single valued function of two variables. This technique accelerates the ray trace process by advancing ray trace starting points. It is most applicable when every pixel is to be ray traced. To take advantage of ray vertical coherence in perspective view generation, pixels must be processed sequentially along lines in the simulation display corresponding to vertical planes.

1.2 Height Field Ray Trace Methods

Accelerations due to tile warping and ray vertical coherence cannot generally be combined because pixels on corners of tiles in the simulation display are not generally associated with vertically coherent rays. But height field ray tracing can also be accelerated by using faster height field ray trace methods. These methods can be used alone or in conjunction with either tile warping or ray vertical coherence to further accelerate the photo-realistic terrain visualization process.

Incremental methods are the standard height field ray trace methods [1-5]. As illustrated in Fig.2, they traverse rays in steps along intersections with height field cell walls, i.e., planes containing rows or columns of height field samples. These methods are intuitive and brute force in the sense that no cells are skipped. A height field resides in RAM during incremental ray tracing.

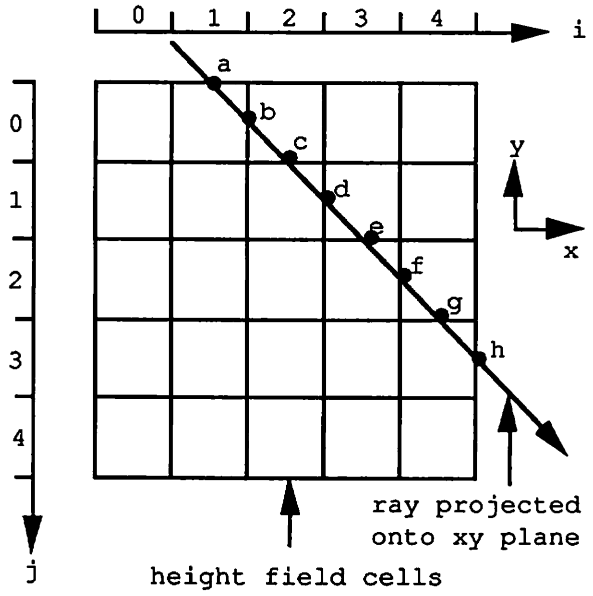


Fig. 2 Incremental ray trace steps a through h.

Hierarchical methods never require more ray trace steps than incremental methods and usually require far fewer steps [6-7]. They typically run faster than incremental methods because they rely on a pre-computed quadtree representation of a series of reduced resolution height fields. This quadtree resides in RAM during ray tracing. In these quadtrees, the height of a quadrant of cells is taken as the height of the tallest cell in that quadrant. As illustrated in Fig.3, hierarchical methods reduce the number of ray trace steps by tracing over entire quadrants of cells whenever minimum ray height over that quadrant exceeds the height of that quadrant. Hierarchical ray tracing proceeds by inspecting quadrants of successively higher resolution. It steps over quadrants that the ray lies completely above and segments quadrants that the ray could potentially pierce into four sub-quadrants. These quadrants are pushed onto a stack of quadtree nodes in reverse order encountered along the ray. They are popped and analyzed in the order encountered until a pierce-point is found or the end of the ray has been reached (i.e., the stack is empty).

Parametric methods are introduced in the next section. They fall into a new class of height field ray trace methods. The height field is horizontally sliced into evenly spaced cross-sections. For each cross-section, the Euclidean distance from each point to the nearest point where the slice cuts through terrain is computed off-line. These planes of distance values are collapsed into *parameter planes* that encode cones of empty space above each height field cell. Cone width is bounded by the terrain relief. Parametric ray tracing occurs along intersections between rays and these cones.

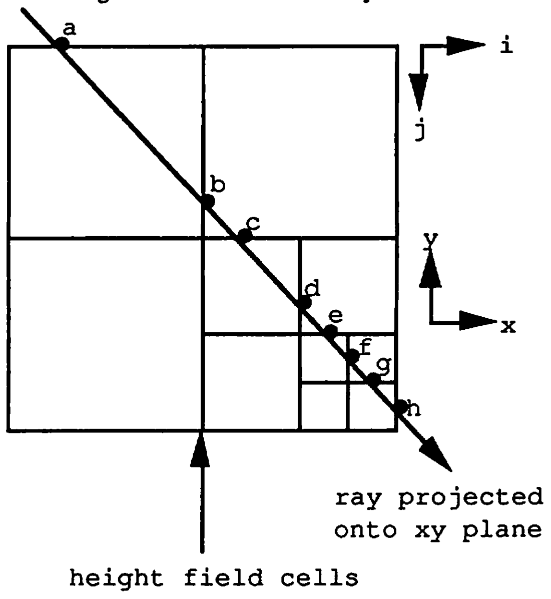


Fig.3 Hierarchical ray trace steps a through h.

2. Parametric Methods

Consider a class of height field ray trace methods in which ray trace steps from any starting point on the ray are bounded by where the ray exits a cone-like volume of empty space balanced on its apex (see Fig.4). The apex is centered on the top of the height field cell that the starting point lies above. At any height, cone width is bounded by the distance to the closest terrain point that high or higher. For each cell, there is a distinct *360° terrain profile* that defines the bounds of its associated cone-like volume. Profile height at a given distance from the apex is the height of the tallest terrain point that ground distance away or closer. Each 360° terrain profile is thus nondecreasing and symmetrical about its associated apex. Parametric ray trace methods refer to height field ray trace methods of this type for which upper bounds on the 360° terrain profiles are represented by a set of curve parameters.

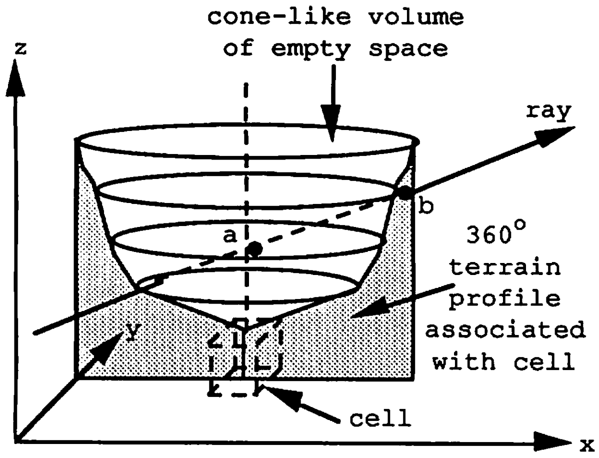


Fig.4 Parametric ray trace step from from a to b.

2.1 Distance Transforms of Height Field Horizontal Cross-Sections

Consider an array $\{z(i,j)\}$ of cell heights, where the heights of the shortest and tallest cells are z_{min} and z_{max}. Specify $K \overset{\Delta}{=}$ the number of height field horizontal cross-sections (where $\overset{\Delta}{=}$ stands for "is defined as"). Define an increasing sequence $\{z_k \quad k = 0,\ldots,K\text{-}1\}$ of uniformly quantized heights from z_{min} to z_{max} as

(1) $z_k = z_{min} + k\Delta_z \qquad k = 0,\ldots,K\text{-}1$

where the uniform height quantization Δ_z is given by

(2) $\Delta_z = (z_{max} - z_{min}) / (K-1)$.

Then for k = 0,...,K-1, the bit planes

(3) $B_k(i,j) = \begin{cases} 1 & z(i,j) \geq z_k \\ 0 & \text{otherwise} \end{cases}$

are the height field horizontal cross-sections at heights z_k. These cross-sections can be used collectively to visualize the distribution of terrain heights. The number of ones in B_k decreases as k increases. Moreover, the pixels of value one in B_{k+1} are a subset of those in B_k and $B_0(i,j) = 1$ for all (i,j).

Distance transforms (*DT's*) of these bit maps are arrays of distances from each pixel in the bit map to the nearest bit map pixel of value one. They contain ground distances, in units of height field cells, from each cell to the closest cell of height no less than the height of the horizontal cross-section. The DT of height field horizontal cross-section k is

(4) $D_k(i,j) \overset{\Delta}{=}$ distance from (i,j) to the closest (i',j') such that $B_k(i',j') = 1$.

Let us refer to $\{D_k(i,j)\}$ as the kth ***height distributional distance transform*** (*HDDT*) plane. Note that for all (i,j), $D_{k+1}(i,j) \geq D_k(i,j)$ and $D_0(i,j) = 0$.

Parametric ray trace methods require exact Euclidean DT's at every pixel. There are several methods for computing Euclidean DT's of bit maps [8-12]. Some produce approximations while others yield exact results. The *unified distance transform algorithm* of [12] was used to generate DT's here because it rapidly computes exact Euclidean DT's of arbitrary bit maps at every pixel even on general purpose computers.

A height field of the McCall Idaho area obtained from the USGS is visualized in Fig.5(a) as a gray-scale intensity field in which high intensity corresponds to high altitude. The height ranges from 1489m to 2270m, the sample spacing is 30m and the height field has 457x323 samples. As indicated in Fig.5(a), the terrain relief in the McCall quadrangle is moderate and most of the highest altitudes occur in the northern portion. A horizontal cross-section of this height field at a height of approximately 1890m is shown in Fig.5(b), where the black regions correspond to where the horizontal plane cuts through the terrain. The complete exact Euclidean distance transform of this height field cross-section is visualized in Fig.5(c) as a gray-scale intensity field in which high intensity corresponds to large distance from black regions in the cross-section.

2.2 Parameter Planes

HDDT planes can be pre-computed for stacks of height field horizontal cross-sections and can conceptually be stored in a data structure called an *HDDT stack*. For each height field cell (i,j), there is a non-decreasing *HDDT sequence* $\{D_k(i,j) \quad k = 0,\ldots,K\text{-}1\}$ of Euclidean distances from cell (i,j) to the nearest cell of height no less than the cross-section heights z_k. In the limit as $\Delta_z \rightarrow 0$, these sequences become continuous non-decreasing mappings of ground distance vs. height, called *HDDT profiles* for cells (i,j). However, HDDT profile values are pre-computed for only certain height quantizations $z=z_k$.

To reduce RAM requirements, suppose that for each cell (i,j), the HDDT sequence $\{D_k(i,j) \quad k = 0,\ldots,K\text{-}1\}$ is replaced by a simple parametric representation that acts

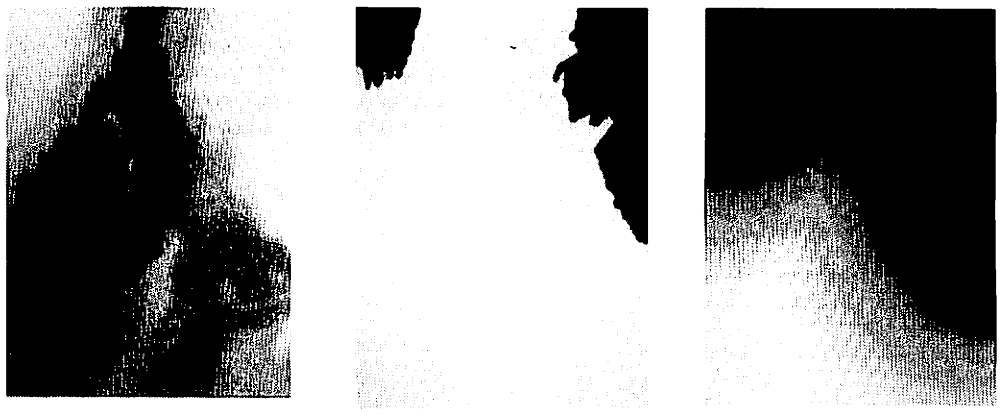


Fig.5 (a) A height field visualized as intensity increasing with altitude (7.5' USGS McCall quadrangle, Idaho). (b) Height field horizontal cross-section at approximately 1890m. (c) Euclidean distance transform of (b).

as a lower bound to the true HDDT profile. Then only the parameters of these representations would need to be stored and read into RAM. Arrays of cell parameter values, called *parameter planes*, would replace the HDDT stack, where each plane corresponds to a different parameter. These parameter planes can be updated off-line each time a new HDDT plane is generated. Once updated, that HDDT plane can be discarded. Parameter planes can thus be based on an arbitrary number of height field cross-sections because no more than one HDDT plane ever needs to reside in RAM at any given time during parameter plane generation.

The simplest parametric representations possible for HDDT profiles are linear lower bounds. These representations have only two parameters, namely a slope and a z intercept. For each cell (i,j), the z intercept must be chosen greater than or equal to (equal to in this paper) the height z_k of the first horizontal cross-section for which $D_k(i,j) \neq 0$. Once the z intercept for cell (i,j) has been specified, the slope for cell (i,j) is updated each time a new HDDT plane is generated. The slope for cell (i,j) must be chosen as the slope of some line that passes through the z intercept for cell (i,j) and a point $(D_k(i,j), z_{k+1})$ for which z_{k+1} exceeds the z intercept. Of all such lines, the line of least slope must be taken as the linear lower bound.

Let $m_p(i,j)$ and $z_p(i,j)$ be the slope and z intercept parameter values associated with cell (i,j). Then $\{m_p(i,j)\}$ and $\{z_p(i,j)\}$ are the slope and intercept parameter planes. These parameters have an interesting physical interpretation. $z_p(i,j)$ can be thought of as the height of the apex of a cone of empty space situated directly above cell (i,j) balanced on its apex. The reciprocal of $m_p(i,j)$ is the slope of a line which, when swept through 360° about the apex, defines a cone surface. In effect, the parameter planes encode conical volumes of empty space situated above each height field cell, where cone width is bounded by the terrain relief. As shown in Fig.6, parametric ray trace steps occur along intersections between rays and surfaces of such cones. Unlike the cone-like volume in Fig.4, these cones have an apex that may float above the associated height field cell and they are parameterized with line slope and intercept parameters. Each cone in Fig.6 would fit inside an associated cone-like volume such as the one depicted in Fig.4.

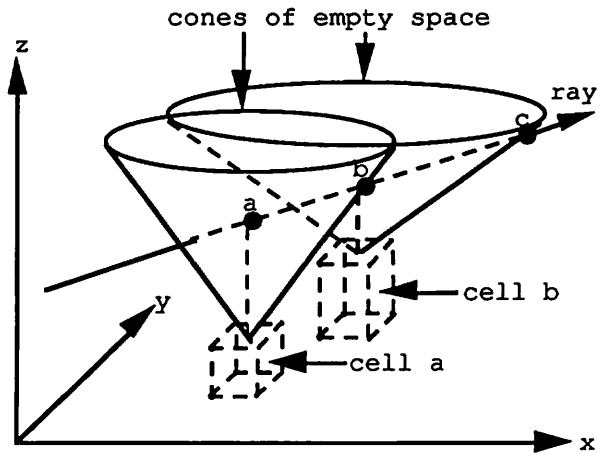


Fig.6 Parametric ray trace steps a through c as intersections between rays and surfaces of cones of empty space.

Incremental ray tracing requires RAM for storage of the height field. Parametric ray tracing requires RAM for storage of the height field and its parameter planes. If all data is stored in 4 byte floating point, the RAM requirements for linear parametric ray tracing are only 3 times greater than those for incremental ray tracing. In contrast, hierarchical ray tracing requires RAM for storage of a quadtree representation of a height field resolution pyramid. The quadtree data structure is more complex than the simple raster data structures required for incremental and parametric ray tracing. Among other things, the quadtree nodes must contain quadrant height, xy quadrant bounds and pointers to four child quadrant nodes. Assuming that each node requires 32 bytes, hierarchical methods require roughly 10 times more RAM than incremental methods.

2.3 Analytical Computation of Parametric Ray Trace Steps

In three-dimensional local planar coordinate systems with x (east), y (north) and z (height) axes, rays can be characterized by a starting point (or viewpoint) $\mathbf{p_s} \overset{\Delta}{=} [x_s,y_s,z_s]$ and a direction vector $\mathbf{d} \overset{\Delta}{=} [d_x,d_y,d_z]$ as

(5) $\mathbf{p} = \mathbf{p_s} + t\,\mathbf{d}, \qquad t \geq 0$

where the variable t specifies distinct points $\mathbf{p} \overset{\Delta}{=} [x,y,z]$ on the ray. $\mathbf{p_s}$ and $\mathbf{d}$ are derived from the parameters of the line-of-sight through some pixel in the simulation display. These parameters are dictated by the math model associated with the simulation. If the simulation model is that of a frame camera, then the simulation is a perspective view so $\mathbf{p_s}$ is camera position and $\mathbf{d}$ is a function of pixel coordinates, camera focal length and camera rotation (tilt, swing, azimuth). At zero rotation, the camera frame is oriented such that its x axis points straight east, its y axis points straight north and its z axis (which is the optical axis pointed in the opposite direction) points straight up. The camera frame is obtained by rotating the zero rotation frame by the azimuth angle (from 0° to 360°) clockwise about its z axis, rotating the resulting frame by the tilt angle (from 0° to 180°) about its x axis and rotating the resulting frame by the swing angle minus 180° (from 0° to 360°) clockwise about its z axis. Physically, azimuth reflects how far off north the camera is pointed. Tilt is the rotation between a ray pointed straight down and the optical axis. The camera points up when tilt exceeds 90° and points down when it is less than 90°. Swing reflects how much the camera is "twisted" once azimuth and tilt have been applied.

For any ray, the xy (ground) distance D traversed, in units of height field cells, from the ray point at height z_r to the ray point at height z is given by the *ray xy traversal line*

(6) $(d_z\Delta)D = d_r(z-z_r)$

where Δ is the distance (in meters) between adjacent height field samples and

(7) $d_r \overset{\Delta}{=} (d_x^2+d_y^2)^{1/2}$.

If $d_z \neq 0$, the ray xy traversal line can be expressed as

(8) $D = m_r(z-z_r)$

(9) $m_r \overset{\Delta}{=} \Delta \cdot d_r/d_z, \quad d_z \neq 0$,

i.e., if $d_z \neq 0$, the ray xy traversal line has slope m_r and intercept z_r. For rays of constant height ($d_z=0$), D takes on all real values at $z=z_r$ but is undefined for all other z. Thus, D is a linear mapping of xy distance vs. ray height traversed. It is decreasing for down-looking rays ($d_z<0$), increasing for up-looking rays ($d_z>0$) and of infinite slope for constant height rays. Moreover, ray xy traversal lines have slopes m_r that become steeper as their rays become more shallow (i.e., as d_z decreases in magnitude).

Let $[x_r,y_r,z_r]$ be a point on the ray. Let D^* be the xy distance associated with the parametric ray trace step from $[x_r,y_r,z_r]$. Let z^* be the height of the point stepped to on the ray. From any point $[x_r,y_r,z_r]$ situated over height field cell (i,j) on the ray, $[z^*,D^*]$ is dictated by how the ray is positioned over the height field and the terrain relief. The ground distance mapping associated with $[x_r,y_r,z_r]$ is dictated by z_r and how the ray is positioned. The HDDT profile associated with cell (i,j) is dictated by the terrain relief. D^* is limited by the distance to the point on the ray at height z^* whose ground distance from $[x_r,y_r,z_r]$ equals the ground distance from $[x_r,y_r,z_r]$ to the closest cell of height no less than z^*. The maximum possible value for D^* consistent with parametric height field ray tracing methods is given by the intersection closest to the z axis between the ray xy traversal line and the HDDT profile.

Parametric methods compute lower bound estimates for these maximum ray trace step lengths as intersections $[z^*, D^*]$ between ray xy traversal lines and parametric representations of HDDT profiles for cells (i,j). When the parametric representations are linear, these intersections are easy to compute analytically (see Fig.7). The linear HDDT profile for cell (i,j) is given by

(10) $D = m_p(i,j)\,[z - z_p(i,j)]$.

The intersection between the linear HDDT profile for cell (i,j) and the ray xy traversal line associated with ray point $[x_r,y_r,z_r]$ that lies above cell (i,j) is

$$(11)\quad z^* = \begin{cases} z_r & d_z=0 \\ \dfrac{m_p(i,j)z_p(i,j)-m_r z_r}{m_p(i,j)-m_r} & d_z\neq 0,\ m_p(i,j)\neq m_r \\ \text{undefined} & d_z\neq 0,\ m_p(i,j)=m_r \end{cases}$$

$$(12)\quad D^* = \begin{cases} m_p(i,j)[z^*-z_p(i,j)] & z^* \text{ defined} \\ \text{undefined} & z^* \text{ undefined} \end{cases}$$

In Fig.7, the solid line is the linear HDDT profile (10) and the four dashed lines are ray xy traversal lines (6) or (8) with slopes $m_r(i)$ $i=1,2,3,4$ that intersect the rays at $[z^*(i), D^*(i)]$ $i=1,2,3$. Ray 1 points down, ray 2 is constant height and rays 3-4 point up. As for rays 1-3, if $D^* > 0$, the parametric ray trace step from the current ray point $[x_r,y_r,z_r]$ is to the ray point $[x^*,y^*,z^*]$ an xy distance D^* away where

$$(13)\quad [x^*,y^*] = \begin{cases} [x_r+(z^*-z_r)d_x/d_z, y_r+(z^*-z_r)d_y/d_z] & d_z\neq 0 \\ [x_r+(D^*\Delta)d_x/d_r, y_r+(D^*\Delta)d_y/d_r] & d_z=0 \end{cases}$$

If D^* is undefined, less than zero or the ray point stepped to ends up above the same cell as the previous ray point, then it is necessary to *jump start* the parametric ray trace process. Jump starting is the process of stepping incrementally to the next cell so that parametric ray tracing can continue.

Furthermore, as for ray 4 in Fig.7, if the ray points up and for $D\geq 0$, the ray xy traversal line associated with the current ray point lies completely beneath the associated linear HDDT profile, then the ray can never pierce the ground. In this case, there is no xy distance from the current ray point for which the tallest terrain point that far away is at least as high as the ray point that distance ahead. In other words, the portion of the ray emanating from the current ray point never pierces the cone associated with the cell that the current ray point lies above. This mechanism for determining that certain rays never pierce the ground without having to ray trace further allows parametric ray trace methods to process certain rays with incredible efficiency. In addition, if the portion of the ray emanating from the current ray point does pierce the cone but the pierce point lies outside the height field bounding box, then the ray never pierces the height field surface and parametric ray tracing is complete.

2.4 Parametric Height Field Ray Trace Algorithm

The parametric height field ray trace algorithm can be summarized as follows:

```
1. Initialize:
   a. if ray does not pierce height field bounding box
         then return
      else $[x_r,y_r,z_r]$ ← ray trace starting point
   b. if ray points straight up or starts under height field
         then return
   c. (i,j) ← index of cell associated with ray trace
               starting point $[x_r,y_r]$
      if ray does not point up then
         $[x_r,y_r,z_r]$ ← point where ray exits cell (i,j)
      if $d_z\neq 0$ then compute $m_r$
      status ← 1

while status = 1
  2. Compute Pierce-Point:
     if ray pierces cell (i,j) then return pierce-point
  3. Determine Next Parametric Ray Trace Step:
     if $z_r \geq z(i,j)$ then
        a. if $d_z > 0$ and $z_r > z_p(i,j)$ and $m_r \leq m_p(i,j)$
              then return
           Compute $[z^*,D^*]$.
           if $D^* > 0$ then $[x_r,y_r,z_r]$ ← $[x^*,y^*,z^*]$
        b. (i',j') ← (i,j)
           (i,j) ← index of cell associated with $[x_r,y_r]$
           if (i,j) out of height field bounds
              then return
  4. Jump Start By Determining Next Incremental Ray
     Trace Step:
     if $z_r < z(i,j)$ or ($z_r \geq z(i,j)$ and i'=i and j'=j) then
        if $d_z > 0$ then
           $[x_r,y_r,z_r]$ ← point where ray exits cell (i,j)
           (i,j) ← index of cell for which $[x_r,y_r,z_r]$ is
                   the ray entry point
           if (i,j) out of height field bounds
              then return
        else
           (i,j) ← index of cell for which $[x_r,y_r,z_r]$ is
                   the ray entry point
           if (i,j) out of height field bounds
              then return
           $[x_r,y_r,z_r]$ ← point where ray exits cell (i,j)
```

To summarize, the ray trace starting point is used as the first current ray point. A pierce-point calculation is required if somewhere over the current cell, ray height drops below the height of that cell. If no pierce-point is found, the next parametric ray trace step is determined. Whenever the ray points up and the ray xy traversal line associated with the current ray point lies completely beneath the associated linear HDDT profile for $D\geq 0$, the ray never pierces the ground. Otherwise, if a parametric ray trace step could not be determined or the ray point stepped to lies in the same cell as the current ray point, it becomes necessary to jump start the ray trace process by determining the next incremental ray trace step. The current ray point is then replaced by the ray point stepped to. If the new current ray point lands outside the height field bounding box, ray tracing is finished. Otherwise, the process iterates so that the next ray trace step can be determined.

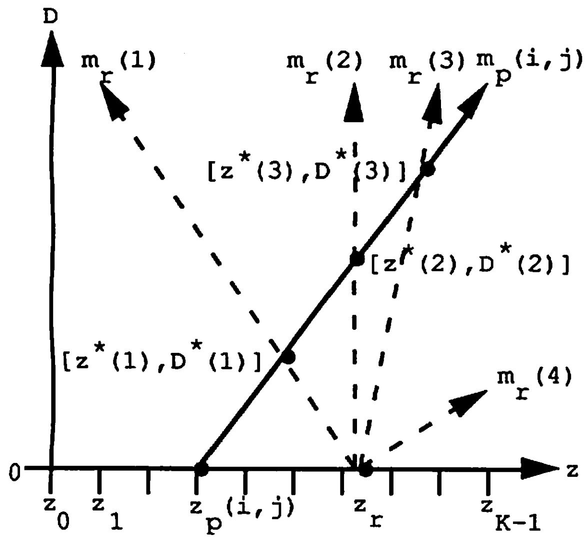


Fig.7 Parametric ray trace steps as intersections between ray xy traversal lines (dashed) and a linear HDDT profile (solid).

3. Experimental Results

The amount of time required to generate photo-realistic views of terrain by height field ray tracing depends on the size and resolution of the height field, the obliqueness of the view, the number of rays traced, the speed or number of ray trace processors and the efficiency of the ray trace process. For height fields of fixed resolution, the number of potential ray trace steps is directly proportional to the height field range in east or north. For height fields of fixed area coverage, the number of potential ray trace steps is directly proportional to height field resolution. Oblique views of terrain can be particularly time consuming to generate by height field ray tracing because shallow rays often traverse long distances over the ground before they pierce terrain. View generation times vary in inverse proportion to the square of the spacing between ray traced pixels in the simulation display since tile warping, which is generally much cheaper than ray tracing, can be used to fill in the gaps. View generation times also vary in direct proportion to the number of ray trace processors, processor speed and the efficiency of the implemented ray trace process. The effect that the height field ray trace method has on ray trace efficiency is investigated in this section. All tests were performed on the same 10 MIPS Sun 4/280 processor.

An experiment was performed with the 16 oblique views, listed in Table 1, of the McCall height field in Fig.5(a) (z_{min}=1489 m, z_{max}=2270 m). An imaginary frame camera with a focal length of 50 mm was devised. The interior orientation was chosen such that each pixel corresponds to 1/4 mm on an imaginary focal plane and pixel (0,0) lies at the center of a 512x512 simulation display. Each view thus has a wide opening angle of about 104°. Well-distributed rays were obtained by processing lines-of-sight through every 16th pixel in each view.

For each view, the number of rays that pierce the terrain (n_{pierce}) out of 1024 is recorded in Table 1. Since all 16 viewpoints lie within the height field bounding box, all rays must be ray traced. The mean numbers of incremental, hierarchical and parametric ray trace steps over all rays traced in all views (i.e., n_I, n_H and n_P) are recorded in Table 1. The amount of time (t) that it takes to ray trace 16 views (every 16th pixel) is recorded in the last row of Table 1. K=160 height field horizontal cross-sections (Δ_z≈5 m) were used to generate the parameter planes. The average number of incremental, hierarchical and parametric ray trace steps ranged from roughly 130 to 275, 6 to 9 and 2 to 6 respectively. The number of incremental to hierarchical ray trace steps averaged roughly 27 to 1. The number of incremental to parametric ray trace steps averaged roughly 56 to 1. The number of hierarchical to parametric ray trace steps averaged roughly 2.1 to 1. The hierarchical method ran roughly 7 times faster than the incremental method. The parametric method ran roughly 40 times faster than the incremental method and roughly 6 times faster than the hierarchical method. Parametric ray trace steps cost somewhat more than incremental ray trace steps but considerably less than hierarchical ray trace steps.

The three ray trace methods can also be compared with respect to their memory requirements. The incremental method required roughly 0.59 Mbytes for the height field (4 bytes per height sample). The hierarchical method required roughly 6.3 Mbytes for the quadtree representation of the height field resolution pyramid (32 bytes per quadtree node). The parametric method required roughly 1.75 Mbytes for the height field and its two parameter planes (4 bytes per parameter value). The ratios of hierarchical to incremental and parametric memory requirements were roughly 10.7 to 1 and 3.5 to 1 respectively. The ratio of parametric to incremental memory requirements was roughly 3 to 1.

Although the number of incremental ray trace steps basically increases in direct proportion to height field resolution, it is theorized that the number of parametric ray trace steps remains relatively constant. If true, the ratio of incremental to parametric ray trace run times would increase in proportion to height field resolution

Table 1: Test View Parameters and Ray Trace Method Performance

view	viewpoint (meters)			orientation (degrees)			n_{pierce} (≤ 1024)	n_I	n_H	n_P
	x_s	y_s	z_s	tilt	swing	azimuth				
1	569400	4969900	1800	86.3	176.7	54.6	518	246.76	6.14	2.94
2	573975	4982200	2200	72.1	180.1	0.0	600	219.26	8.23	3.89
3	573975	4982200	1650	112.2	180.1	0.0	345	262.32	8.39	3.72
4	569250	4976235	2000	57.3	180.2	89.9	748	164.34	9.34	4.13
5	569250	4976235	2000	80.2	180.1	90.0	557	240.21	8.00	3.59
6	578700	4976235	2200	57.3	179.8	-89.9	717	163.54	9.48	5.99
7	578000	4974000	1650	83.2	180.8	-56.2	555	195.83	6.69	4.24
8	569230	4975380	1800	82.5	179.8	76.4	552	229.33	8.53	3.27
9	573500	4982000	1700	78.1	179.1	-11.7	584	196.40	8.37	4.02
10	572800	4975600	2200	76.9	174.0	81.6	571	179.37	7.98	3.00
11	578500	4974000	1700	86.5	183.2	-57.5	538	212.71	8.96	3.81
12	569400	4969900	2200	73.8	180.2	48.3	584	248.08	6.52	2.91
13	578500	4970000	2200	53.2	186.4	-44.8	760	172.19	8.16	4.67
14	569100	4976235	2000	91.7	180.7	89.8	474	273.70	7.27	3.11
15	574200	4981000	1600	119.8	182.1	-76.0	302	218.09	6.22	3.34
16	573400	4981000	1600	91.8	181.6	86.4	521	130.08	7.84	3.28
mean							557.88	209.51	7.88	3.74
t (sec)								350	51	9

and the benefits of parametric ray tracing would be augmented. Work is currently under way to study the effect that data field resolution has on the parametric ray trace process.

4. Conclusions

Parametric ray trace methods often generate oblique photo-realistic views of terrain much more rapidly than hierarchical methods, which often generate such views much more rapidly than incremental methods. The number of ray trace steps attributable to parametric and hierarchical methods can theoretically never exceed the number attributable to incremental methods.

Parametric and hierarchical methods require fewer ray trace steps than incremental methods because they use results of height field pre-processing that require additional RAM. Incremental methods require RAM for one height field. Hierarchical methods require RAM for a quadtree representation of a height field resolution pyramid. Linear parametric methods require RAM for one height field and two parameter planes. Linear parametric methods require roughly 3 times more RAM than incremental methods. Hierarchical methods require roughly 10 times more RAM than incremental methods. However, the hierarchical data structure is more complex than the simple raster data structures associated with incremental and parametric ray tracing.

Acknowledgements

The authors wish to thank Dr. Walt Eppler for the time he spent reviewing this manuscript.

References

1. Unruh, J. and Mikhail, E., "Image Simulation from Digital Data", ACSM Fall Tech. Meet., (1977), 1-12.
2. Dungan, W. Jr., "A Terrain and Cloud Computer Image Generation Model", Computer Graphics, vol.13, no.2, (1979), 143-150.
3. Coquillart, S. and Gangnet, M., "Shaded Display of Digital Maps", IEEE Comput. Graph. Appl., (July 1984), 35-42.
4. Anderson, D. P., "Hidden Line Elimination in Projected Grid Surfaces", ACM Trans. Graphics, vol.1, no.4, (October 1982), 274-288.
5. Musgrave, F. K., "Grid Tracing: Fast Ray Tracing for Height Fields", YALEU/DCS/RR-639, (July 1988).
6. Kajiya, J. T., "New Techniques for Ray Tracing Procedurally Defined Objects", ACM Transactions on Graphics, vol.2, no.3, (July 1983), 161-181.
7. Mastin, G. A., Watterberg, P. A. and Mareda, J. F., "Fourier Synthesis of Ocean Scenes", IEEE Comput. Graph. Appl., vol.3, (March 1987), 16-23.
8. Rosenfeld, A. and Pfaltz, J. L., "Sequential Operations in Digital Picture Processing", J. Assoc. Comput. Mach., vol.13, (1966), 471-494.
9. Danielsson, E., "Euclidean Distance Mapping", Comp. Graph. Img. Proc., vol.14, (1980), 227-248.
10. Yamada, H., "Complete Euclidean Distance Transform by Parallel Operation", Proc. 7th Int. Conf. Pat. Recog. (Montreal Canada, 1984), 69-71.
11. Borgefors, G., "Distance Transforms in Digital Images", Computer Vision, Graphics and Image Processing, vol.34, (1986), 344-371.
12. Paglieroni, D. W., "A Unified Distance Transform Algorithm and Architecture", Machine Vision and Applications, vol.5, (1992), 47-55.

Partitioning Tree Image Representation and Generation from 3D Geometric Models

Bruce F. Naylor

AT&T Bell Laboratories
Murray Hill, NJ 07974
naylor@research.att.com

Abstract

While almost all research on image representation has assumed an underlying discrete space, the most common sources of images have the structure of the continuum. Although employing discrete space representations leads to simple algorithms, among its costs are quantization errors, significant verbosity and lack of structural information. A neglected alternative is the use of continuous space representations. In this paper we discuss one such representation and algorithms for its generation from views of 3D continuous space geometric models. For this we use <u>*binary space partitioning trees*</u> *for representing both the model and the image. Our approach falls under the general rubric of visible surface algorithms, providing an object-space algorithm which under certain conditions requires only sub-linear time for a partitioning tree represented model, and in general exploits occlusion so that the computational cost converges toward the complexity of the image as the depth complexity increases. Visible edges can also be generated as a step following visible surface determination. However, an important contextual difference is that the resulting image trees are used in subsequent continuous space operations. These include affine transformations, set operations, and metric calculations, which can be used to provide image compositing, incremental image modification in a sequence of frames, and facilitating matching for computer vision/robotics. Image trees can also be used with the hemicube and light buffer illumination methods as a replacement for regular grids, thereby providing exact rather than approximate visibility.*

Discrete vs. Continuous Space

We have come to think of images as synonymous with a 2D array of pixels. However, this is an artifact of the transducers we use to convert between the physical domain and the informational domain. Physical space at the resolution with which we are concerned is most effectively modeled mathematically as being continuous, that is, as having the structure of the Reals (or at least the Rationals) as opposed to the structure of the Integers. Modeling space as being defined on a regular lattice, while simple, is verbose and induces quantization which reduces accuracy and can introduce visible artifacts. Using nothing other than a lattice for the representation provides no image dependent structure such as edges.

Consider applying to a discrete image an affine transformation, an elementary spatial operation. The solution for this is developed by reasoning not merely in discrete space but in the continuous domain as well: samples are used to reconstruct a "virtual" continuous function which is then resampled. However, the quantization effects can become rather apparent should the transform entail a significant increase in size and a rotation by some small angle, despite the use of high quality filters. This is due to such factors as ringing, blurring, aliasing, and anisotropic effects which cannot all be simultaneously minimized (see, for example, [Mitchell and Netravali 88]). More importantly, discontinuities become increasingly smeared as one increases the size, since the convolution assumes a band-limited signal, i.e. an image with no edges. This has practical implications when texture mapping is used to define the color of surfaces in 3D: since a texture map can be enlarged arbitrarily, a brick texture, for example, will become diffuse instead of exhibiting distinctly separate bricks.

Now consider applying affine transformations to images represented by quadtrees, a spatial structure, developed within the context of a finite discrete space, for reducing verbosity and inducing structure on an image. The algorithm for constructing the new quadtree of the transformed image seems relatively complicated when compared to the corresponding algorithms for continuous space representations: it must resample each transformed leaf node and construct an entirely new tree. In contrast, boundary representations, simplical decompositions, or binary space partitioning trees only require transforming points and/or hyperplanes (a vector-matrix product), and no structural changes are required. An extremal example of this difference is the quad-tree representation of a square occupying a quadrant, which requires 5 nodes, but when slightly rotated or translated the number of nodes is on the order of the number of pixels lying on its boundary (say about 4k for a 1k x 1k grid). This rather dramatic metamorphosis illustrates quite clearly that the quadtree reflects the nature of a finite discrete space, a nature differing from that of the continuum, and that applying arbitrary affine transformations in discrete space can affect the structure of the representation, introducing quantization noise and requiring more complicated algorithms.

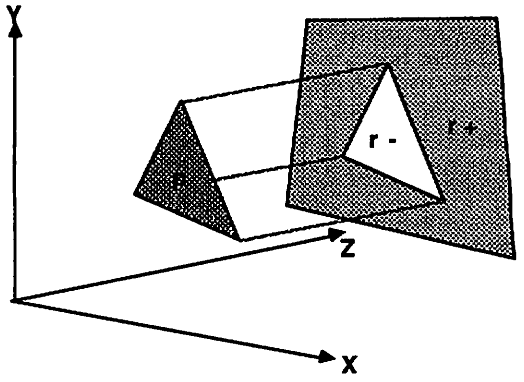
We are inclined to state a stronger proposition: discrete space, as a regular lattice, supports weakly the semantics of the continuum. Assuming this, the difficulties with transforming pixel arrays and quadtrees is not so unexpected. A good model for images is one that treats them as functions mapping a continuous 2D domain to a color space (the 2D domain may be unbounded). Discrete space representations are then treated as approximations of this function, or as evaluations achieved by point-sampling the domain, and discrete space operations are then constructed as approximations to their continuous space analogs. To display the image, conversion to a discrete representation would still be needed, but this now becomes strictly an issue of sampling the image function. (This argument should not be confused with the random vs. raster scan distinction, which is a question of transducer technology, not of computational technology.) With this said, we will now consider methods of generating continuous space image representations from 3D continuous space geometric models.

Visible Surface Algorithms

The context in which continuous space image representations are most easily produced is synthetic image generation. Here one begins with a 3D geometric model, defined using continuous space methods, from which a continuous space image representation is generated. This idea appeared very early in the development of visible surface algorithms, and in [Sutherland, Sproull and Schumacker 74] such algorithms were called *object-space* algorithms. But the approach has been neglected in favor of solutions utilizing quantized spaces (except in the Computational Geometry community).

The algorithm of [Weiler and Atherton 77] is a well known example of a continuous space method for generating images, and since it resembles closely our own approach, we will describe it in some detail. The algorithm operates on a set of polygons defined in a 3D post-perspective screen-space; thus, all projectors are parallel to the z-axis. Each polygon is represented by a boundary representation of some variety. Presumably the polygons are the faces of a collection of polyhedra, but this property is not relied on. The algorithm proceeds by recursively partitioning space until homogeneous regions of the image are generated. Homogeneity in this case means, in 2-space, a region in which only one polygon is visible, or in 3-space, a region which is entirely visible or entirely occluded. The output of the algorithm is a set of polygons in 2-space with disjoint interiors whose union forms the image. These polygons are in general non-convex and contain holes.

At each point in the recursion, a region **r** of space is partitioned into two sub-regions, which we denote as $\mathbf{r}^-$ and $\mathbf{r}^+$. The partitioning set used is a 3-space polygonal cylinder determined by the boundary of a polygon **p**, chosen from among those polygons that intersect **r** (Figure 1). The faces of the cylinder are orthogonal to the xy-plane, and so contain those projectors which go through the boundary of **p**. Since **p** may be of any genus, the sub-regions created by partitioning with **p** are not necessarily connected and are rarely convex. All polygons, including **p**, are then partitioned into subsets lying in $\mathbf{r}^-$ and $\mathbf{r}^+$, where we take $\mathbf{r}^-$ to be the sub-region containing **p**, i.e. the interior sub-region, and $\mathbf{r}^+$ to be the exterior sub-region.

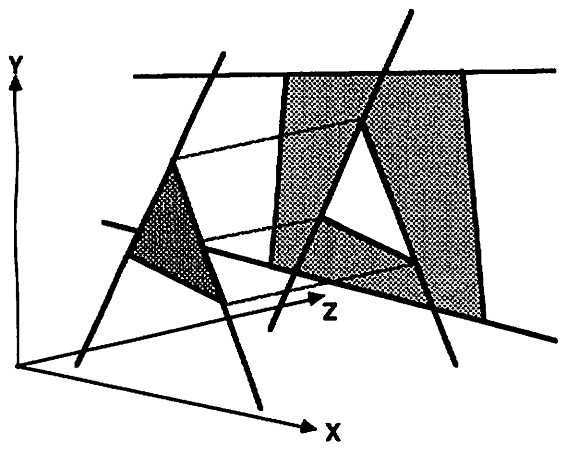


Weiler-Atherton Algorithm
Figure 1

Whenever there is some polygon **p'** in **r** with supporting hyperplane **h** such that $\mathbf{p}' = \mathbf{h} \cap \mathbf{r}$, then all polygons lying "behind" **p'** are occluded by **p'**; such a polygon was called a <u>surrounder</u> in the literature on visible surface algorithms of the 70's, taken from the analogy of a 2D window being surrounded by a polygon. The algorithm selects whenever possible the plane of such a polygon as the partitioning set and then treats the "far region" of **p'** as homogeneous, i.e. as being totally occluded, and so terminates recursion in that region and discards the occluded polygons. Note that the cylindrical partitioning by **p** above results in **p** being a surrounder for $\mathbf{r}^-$. Finally, whenever a 3-space region is generated containing no polygons, this region is necessarily homogeneous.

While the algorithm as just described is all that is required to generate a set of polygons forming the image, we have said nothing about which polygon is chosen as the partitioner when no surrounder is present. A typical technique aimed at improving the performance is to initially sort the polygons in z using for each polygon the smallest z-coordinate from among its vertices, and then to maintain this ordering when polygons are partitioned. The partitioner in the absence of a surrounder is then the first in this ordering among those intersecting a region. In the presence of multiple surrounders, the closest one is chosen.

A similar but lesser known approach was described in an unpublished paper by Ivan Sutherland [Sutherland 73], where he develops a visible surface algorithm inspired by the ideas used in the 1D sorting algorithm *quicksort*. Its output is the same as the above algorithm, i.e. a set of disjoint polygons, and it differs from that algorithm primarily in one aspect: in the absence of a surrounder, the partitioning set is a plane through only one edge of a polygon in **r**. The plane then is orthogonal to the xy-plane, or equivalently, it contains the edge and the center of projection (Figure 2). Selecting which edge to use at each point in the recursion is a heuristic process. Sutherland tried several heuristics without reaching any firm conclusions about what method was best. It is interesting to note that Sutherland's paper also contains a section discussing how this algorithm can be used for shadow generation, transparency and collision detection.



Partitioning tree algorithm
Figure 2

As pointed out in [Harp 86], this algorithm can be treated as a binary space partitioning tree algorithm in that it uses a recursive partitioning by arbitrary hyperplanes; however, it does not generate a tree explicitly, a crucial distinction. This is not surprising given the original inspiration, quicksort. For indeed quicksort can be seen as implicitly constructing a 1D binary search tree, which in turn can be interpreted as a 1D partitioning tree. There is a somewhat subtle but important difference however: sorting has been developed in terms of points whereas space partitioning is in terms of hyperplanes. In 1D, and only in 1D, points and hyperplanes have the same dimension, viz. 0, and so it is easy to confuse them. But hyperplanes are (d-1)-dimensional not 0D sets, as are points. And they have an orientation that distinguishes the two halfspaces induced on a d-space, an orientation that can be used for ordering. Points have no such orientation, nor do they partition space, and so cannot be used to order d-space, d>1. This is one way to see why sorting algorithms are not applicable in dimensions other than 1D. Indeed, if we "attach" the ordering relationship to a 1D point, we then have a 1D hyperplane.

It seems apropos before leaving this section to discuss briefly the visible surface algorithm by John Warnock [Warnock 69]. It was the first recursive space partitioning, visible surface algorithm, and it follows the general scenario outlined above, the main difference being that the partitioning hyperplanes are not determined by polygonal edges (also, polygons were not explicitly partitioned). Today we see it as using a quadtree partitioning scheme. However, like the Sutherland algorithm, no explicit tree representation is generated; its output is a set of visible squares typically drawn directly into a pixel array. It is in effect a discrete space solution (also called a *screen-space* algorithm). It was to a certain degree the verbosity of this discrete solution that motivated the development of the two previously described continuous space algorithms.

Partitioning Trees

The binary space partitioning tree was originally developed in the context of visible surface determination. (The appendix contains a summary for those unfamiliar with the method.) [Schumacker et al 69] developed an incipient version that involved manual creation of a binary tree of vertical separating planes so that each object was separated from all other objects in the scene. The tree could then be used to generate a view-dependent visibility priority ordering. In [Fuchs, Kedem and Naylor 80] and [Naylor 81] three advancements were made: 1) the objects themselves were represented by the tree, 2) tree generation was automatic, 3) a dimension independent representation of space was introduced along with the name "binary space partitioning tree". As noted above, Sutherland also developed a number of ideas using this approach without generating a tree, the lack of which presumably contributed to his not realizing the connection between his work and that of [Schumacker et al 69].

A generalized view of partitioning trees sees them not simply as representations of polytopes but as a representation of functions whose domain and range are continuous spaces of finite dimensions d_1 and d_2 respectively: $f : X \in S^{d_1} \Rightarrow Y \in S^{d_2}$. The partitioning tree partitions the domain into a hierarchical collection of sub-domains. Within each sub-domain a value-continuous function f_i defines the value of f within that sub-domain (typically, f_i is defined for all of S^{d_1} as well, although this is not essential). All points in S^{d_1} at which f is value-discontinuous are contained within partitioning hyperplanes. This interpretation is relevant to the work here since images are functions from 2-space to some color space.

A partitioning tree also provides a structure enabling a hierarchical representation of f. As an example, consider polytopes. At the cells of the partitioning, each f_i is a boolean valued constant function indicating whether or not the cell is in the set. The polytope is the set of points **P = closure({ c_i | c_i is an in-cell})**. Thus f is the characteristic function f_χ for the set P; the algorithm for computing f is the point classification algorithm given in [Naylor 81] [Thibault and Naylor 87]. A useful hierarchical representation of f_χ can be obtained by associating with each region **r** a constant function providing the conditional probability of a point being in **P** given that it is known to lie somewhere within **r**. Thus, for example, the value at the root of the tree is the expected value of the function. We use this idea below to detect regions of the image plane that are discovered to be totally occluded yet inhomogeneous.

Partitioning Tree Visible Surface Algorithm

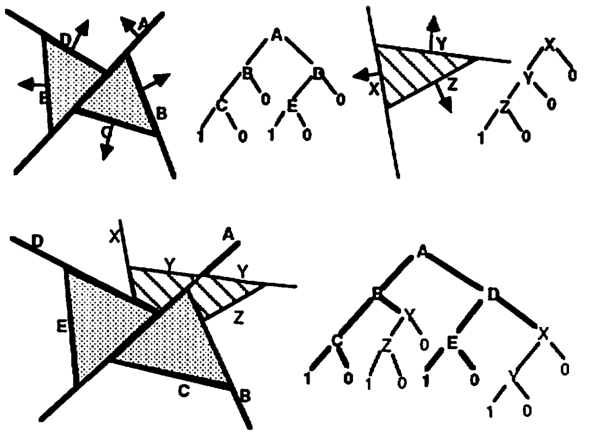
Consider a 3D geometric modeling system in which all geometric sets are represented by partitioning trees. An explicit representation of the model can be formed by taking the union of all the objects comprising the model, resulting in a single tree. This model-tree can then be used, along with a particular view of the model, to generate a total visibility priority ordering on the components of the tree. This ordering can be either far-to-near (back-to-front) or near-to-far (front-to-back). Generating this ordering can be combined with view volume clipping, which performs a non-destructive intersection operation between the model and the view volume, generally in sub-linear time [Naylor 90b].

Regardless of the ordering, a partitioning tree representing the image can be generated by forming the union of the faces in priority order. More specifically, consider a near-to-far ordering with the initial value of the image being the empty set.

1) project each face onto the 2D projection plane and let the attributes of each face be its color.

2) form the 2-space union of the faces in priority order:
   image = Union_Sets(image, face)
   where the attributes of the image take precedent over those of the face.

Thus faces are projected to 2D regions of the image plane, and the effect of higher priority faces occluding lower priority faces is achieved by letting the attributes of the image tree take precedence over those of the current face being "added" to the image. If the faces are represented by partitioning trees, then the union can be performed using tree merging [Naylor, Amanatides and Thibault 90] (figure 3), or if by b-reps, then by the algorithm given in [Thibault and Naylor 87]. If the reverse ordering is used (far-to-near), then the attributes of the new face would take precedence over those of the image tree. We see then that visible surface problem can be reduced to ordered set operations on polyhedral faces. (All of what has been said for 3D -> 2D holds for any d > 1, since partitioning trees and their algorithms are dimension independent.)



A union operation between two faces with attribute precedence
Figure 3

With either ordering, non-refractive transparency can be supported by using a "merge attributes" method that blends colors according to their opacity (alpha values). Given two polygons p_1 and p_2, in which p_1 has color c_1 and opacity α_1 and occludes p_2 which has color c_2 and opacity α_2, then the resulting color is $c_{1,2} = (c_1 * \alpha_1) + (c_2 * \alpha_2) * (1-\alpha_1)$ and opacity is $\alpha_{1,2} = \alpha_1 + (1-\alpha_1) * \alpha_2$ [Porter and Duff 84].

In addition the set of visible edges can be generated, if desired, by performing a closure operation which determines for each sub-hyperplane which subsets have a heterogeneous neighborhood [Naylor, Amanatides and Thibault

90]. So for example, in Figure 3 the subsets of hyperplane A that have homogeneous neighborhoods and so would not be on a discontinuity in the image are those that separate the two polka-dotted cells or two out-cells. This then provides a continuous space visible edge (hidden line) algorithm as an additional step after the visible surface algorithm.

A 3D variant is obtained by transforming the faces into 3D post-perspective screen-space. Then each face is considered to define a polygonal cylinder as in [Weiler and Atherton 77]. The union operation is now on 3D cylinders bounded on the near side by the plane of the face (see Figure 2). The faces could then be added to the image tree in any order, although near-to-far still has advantages as discussed below. Note that the 3D image tree that this produces represents in continuous space exactly the same function represented in discrete space by the standard {frame-buffer, z-buffer} structure.

The algorithm can also be performed in model-space, in which the cylinders are instead cones whose conical-vertex is the center of projection. This then becomes the algorithm present in [Chin and Feiner 89] which they applied to shadow generation, instead of image/visible-surface generation[1]. The resulting model-space image tree can then be transformed by the viewing transformation into screen-space. Working in model-space is preferable numerically, as it avoids the problems encountered as a consequence of the non-linear perspective projection which com-presses the depth at rate of z^{-2}. Note that this problem can be ameliorated somewhat by attempting to match the distribution created by the projection to the distribution of floating-point representable numbers. Uniformly distributed points in model-space become more compressed by the perspective projection the greater the depth. Floating-point representable numbers become more dense the closer the value is to 0. The standard mapping of the near plane to z=0 and the far plane to z=1 results in a mismatch: the greater the model-space depth the further the projected depth-value is from 0. This is trivially rectified by mapping the far plane to 0, and the near plane to -1 if in a left-handed system or to +1 if in a right-handed system.

Now let us compare our algorithm, using a near-to-far ordering, both to the Weiler and Atherton algorithm and to the Sutherland algorithm. They are of course all quite similar. They recursively partition space using at each stage a binary partitioning set (i.e. any (d-1)-set that partitions a d-region into two d-regions), and the partitioning is determined by planes containing either a polyhedral edge and the center of projection and/or by planes of faces. Aside from differences arising from the availability of a priority ordering (to be discussed below), the relationship of our algorithm to Sutherland's is simple: the order of "edge selection", i.e. partitioning hyperplane selection, is pre-determined by the priority ordering of the faces and the tree representing each face. And of course, we explicitly construct a tree to represent the output.

The primary difference between our method and that of Weiler and Atherton, once again other than the priority ordering, is the representation of polygons: their representation is a variety of b-reps while ours is partitioning trees. This difference manifests both in the algorithms for set operation (between faces), and the form of the output (a graph vs. a tree). It is our contention that the set operation algorithm for b-reps are more complicated, slower, and less numerically robust than the corresponding algorithm for partitioning trees. Some early indication of this is suggested by the fact that the original set operation algorithm given in [Weiler and Atherton 77], which is based on a kind of parity counting of intersections, fails to handle co-incident boundaries correctly. A correct but more involved solution was presented later in [Weiler 80] based on Euler operations.

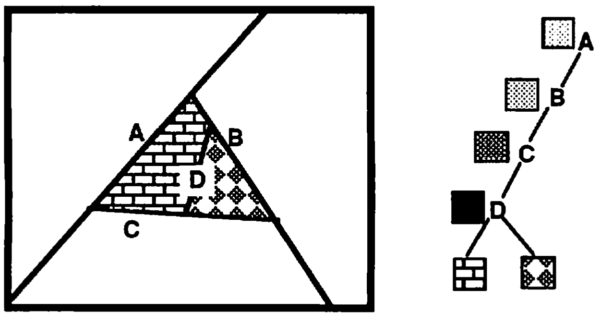
If in the case of partitioning trees, the boundary is already represented by 2D partitioning trees lying in a 3D hyperplane, as discussed in [Naylor 90a], then their representation in screen space either as 2-space entities or as 3-space cylinders is trivial, requiring the application of a single affine transformation (the inverse of the viewing transformation used for points). The tree merging algorithm can then be used to form the image tree as shown in Figure 1 above. Note that a single 2D partitioning tree can represent multiple coplanar connected components (faces), and a well built tree will generally yield better performance than operations on a list of connected components.

The second and more important difference arises from our use of a 3D partitioning tree to represent the model and so to generate a priority ordering. As a consequence, the algorithm is simplified by eliminating the code and execution time for the initial approximating depth sort, and everything associated with the notion of surrounders (detection and ordering). Of greater consequence is that the partitioning of faces by any

[1] Both their ideas and our ideas on this subject occurred independently. We first realized the potential presented here during the period in which we were developing the thesis that partitioning trees could provide a representation of polytopes [Thibault and Naylor 87]. Being able to solve analytically the visible-surface/shadow problems with partitioning trees, analogous to [Sutherland 73], provided part of the supporting evidence for this thesis. [Chin and Feiner 89] extended the ideas in [Thibault and Naylor 87] to generation of shadows. Concurrent with their work, we developed set operations on partitioning trees [Naylor, Amanatides and Thibault 90], which then enabled us to implement the work described in this paper. However, we had originally conceived of our solutions in terms of screen-space using 2D trees, rather than the model-space approach with 3D trees in [Chin and Feiner 89].

occluded subset of an edge is automatically eliminated simply by using a near-to-far ordering. Indeed, at any point in the construction, the image tree can be interpreted as a 2D-polytope representing a *visibility mask*: interior regions correspond to occluded regions and exterior to unoccluded regions. Since additions are made only to unoccluded regions, any intersection between two occluded edges is never computed.

Exploiting the creation of occluded regions of the image plane to reduce computation can be enhanced further by maintaining at internal regions a membership attribute indicating opacity within any region **r**. This can be either the percentage of **r** that is opaque, i.e. the expected value of a point lying in **r** being occluded (see Figure 4), or simply a boolean variable indicating whether **r** is fully occluded or not. Maintaining this membership value during the insertion of a new face amounts to the standard condensation of homogeneous regions used in set operations, the difference being that a region which is homogeneous only with respect to opacity but not color is not replaced by a leaf node. Thus, whenever an internal region becomes fully opaque, it will become a cell of the visibility mask even though a subtree remains defining the image within this fully occluded region. Consequently, this subtree is never again accessed during subsequent processing of lower priority faces. Moreover, when the root region becomes occluded, rendering ceases.



Maintaining % occluded at regions
Figure 4

This captures very simply the ideas present in other work using such masks[2] which require algorithms comparable to set operations on b-reps, and it is the continuous space correlate of pixel masks, be they 1-bit per pixel or many bits per pixel masks (i.e. sub-pixel masks) [Fiume and Fournier 83] [Carpenter 84]. When combined with view-volume clipping, an effect is achieved somewhat analogous to the culling methods presented in [Teller and Sequin 91]. While one would presume that their additional preprocessing would lead to noticeably less computation to generate an image, our method permits a dynamic geometric model (and of course none of the requisite preprocessing and storage of the resulting information).

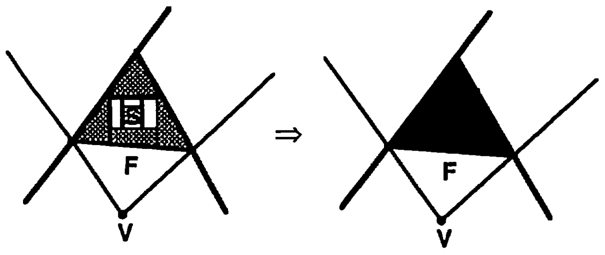
There is, however, a notable deficiency with our scheme as outline above: the order in which the image plane is partitioned is predetermined by the visibility ordering. However, the order in which hyperplanes are chosen affects significantly the "goodness" of the trees, i.e. the efficiency of the search structure provided by the tree. We have come to realize that an efficient partitioning tree is one that represents the set/function as something analogous to a sequence of approximations [Naylor 92]. We have implemented tree construction methods employing expected case models for various elementary operations and these methods produce such trees. What we would then like is to reflect within the image trees this effort at constructing good trees. To achieve this, instead of building the image tree from scratch, we modify (a copy of) the existing model tree so that it will become a representation in 3-space of the occluded and unocculded regions. This can be performed equally well in either model-space or screen-space, with the afore mentioned caveat that screen-space induces a numerically undesirable compression of the depth.

There are several ways to apply this idea; we will describe here only the simplest. We still traverse the tree in a near-to-far priority order. However, after forming the 3D face-beam, instead of performing **image** $\cup$ **face-beam** we perform **model** $\cup$ **face-beam**. As a consequence, entirely occluded faces will be removed by this union operation, and so will not have their face-beam constructed only to find that it is totally occluded, as will occur with the previous method. Indeed, every subtree of the model-tree that is found to be totally occluded will be condensed automatically by the union operation to a single cell <u>before</u> it is encountered in the priority traversal (Figure 5).

An extremal illustration of the power of this approach occurs when an entire object is occluded by a single face of another object. The beam for that face will "engulf" the object, and it will be reduced to a single "occluded" cell. Given our tree construction methods, the computation required in such a case is comparable to computing the union of the beam with a bounding volume of the object, yielding constant time elimination of the occluded object. Thus, under favorable conditions, the visible

[2] A recent example of this is [Sharir and Overmars 92], which is similar in many ways to our method, although it apparently was not implemented. They assume the existence of a visibility priority ordering, maintain a visibility map (our image tree) and a separate mask (our opacity attribute in the image tree). They also rely on merging of these. However, instead of adding one face at a time, they construct a separate visibility map for the next several faces, and then merge this with the "current" map, improving the worst case performance. This idea, if shown to be fruitful, can be easily applied to our method, since there is no algorithmic difference between a tree for a single face and another temporary image tree.

surface can be computed in sub-linear time (and this is in addition to the typically sub-linear clipping of the model to the view volume). More generally, this approach exploits during "beam insertion" the efficient search structures previously generated for each object and the gains from condensing homogeneous regions. Equally important, it retains a desirable residue of this structure in the resulting image tree, and this residue is important for efficient execution of any subsequent spatial operations, such as those discussed in the next section.



With viewer V, subtree S is occluded by face F and is removed by condensation.

Figure 5

Utilizing Image Trees

Given an image tree, one can sample it for display. There are a number of ways to do this. The simplest but most expensive would be to use point classification for each pixel to determine its color. This would, however, allow one to use non-uniform sampling techniques [Mitchell 87] for anti-aliasing. A more reasonable alternative would be to classify scan-lines. But since parametric representations are ideal for scan-conversion, and b-reps are in effect such representations, one can use the algorithm in [Thibault and Naylor 87] to classify an initial b-rep polygon corresponding to the viewport. This yields a disjoint set of convex polygons each as list of vertices, one for each cell, whose attributes are the color of the corresponding cell. Finally, if the faces of the polyhedra are given as b-reps, then these can be retained in the process that constructs the image tree, i.e. during the union operations, as described in [Naylor, Amanatides and Thibault 90]. Thus, by extracting these from the image tree, one obtains an output similar to that generated by the b-rep based algorithms, viz. a set of convex polygons each represented by a list of vertices.

Since only the visible surfaces are scan-converted, texture mapping and per-pixel illumination calculations (Phong shading) will be computed only for visible pixels. In addition, transparency is calculated between polygons rather than repeatedly for each pixel, and so can be provided on systems that do not have the requisite pixel-level hardware. The accuracy of anti-aliasing can be improved significantly, since the visible surface is represented at the resolution provided by floating point, which provides a much higher degree of accuracy than is practical with discrete space. Polygonal edges can be filtered using either continuous or discrete space representations of the filter, with the results being accumulated in the fame buffer using calculations analogous to those described for transparency. This then provides the continuous space version of sub-pixel mask techniques for anti-aliasing presented in [Fiume and Fournier 83] [Carpenter 84] and [Abram, Westover and Whitted 85], and no per-pixel list of micro-polygons with an approximating depth-sort is needed as in [Carpenter 84]. It is also a more efficient form of the per-pixel "analytic" approach in [Catmull 78] which relied on Sutherland's algorithm for visible surfaces. And for line drawings on B&W printers and displays, the visible edges can be used.

As discussed in the introduction, an immediate advantage of continuous space representations is that affine transformations can be applied with ease. Images can be scaled by (S_x, S_y, S_z) corresponding to a model-space scaling of (S_x, S_y, $1/S_z$). A rotation of an image about the screen-space z-axis by θ is comparable to a rotation by θ about the model-space image of this axis, which is the axis through the center of projection and orthogonal to the projection plane. A translation of the image by (T_x, T_y) is equivalent to a shearing with respect to this same model-space axis by ($Sh_x = T_x$, $Sh_y = T_y$). If the perspective is not too severe, then this approximates a similar translation in model-space. For defining texture on a surface, a 2D image tree can be affinely transformed in order to map it into screen-space and then sampled (note that transforming hyperplanes into screen-space requires no "perspective division", but only an affine transformation). In either case, no quantization artifacts, such as enlarged pixels or blurred edges, occur.

Image trees can also be used in subsequent continuous space operations. As noted above, this provides a continuous space version of what has been represented in discrete space by a rgbaz buffer. Therefore, compositing operations can be performed on 3D images as discussed in [Duff 85] (in that work, the space is discrete). These operations can be interpreted as set operations with blending. More specifically we have the following equality:

image(A <set op> B) =
image(A) <set op> image(B), <set op> $\in \{\cup, \cap\}$

And set difference can be used for masking. Since our 3D image trees are of the same data type as any other of our geometric sets, the previously developed set operations can give us compositing immediately. Blending is provided by the same mechanism that provides non-refractive transparency. Using 3D instead of 2D images frees the compositing from being simply a layering of images on top of each other, as in the case for traditional cell animation or video games; i.e. visibility is not restricted to a total order on the individual images, instead they may be interleaved.

While compositing has not been associated with interactive 3D graphics, consider the rather likely situation in which a user has a model comprised of a collection of objects. Typically, the user will engage in modification of only one object at any given time while the view remains stationary. Then an image tree can be constructed for the static objects once at the beginning of this interaction, and the image of the model is generated by

image(model) = image(static-objects) $\cup$ **image(dynamic-object)**

This will yield more benefits the greater the number of static objects, the greater the amount of occlusion, and the greater the duration between selecting a new dynamic object. (For those readers familiar with random-scan display systems, each image tree is analogous to a segment.)

Additionally, it is possible to redraw into a frame-buffer only those faces whose visibility has changed between successive frames. To do this, one needs to maintain in the static-object's image tree a frame index at each node v. This will indicate the last frame in which the subtree rooted at v was changed by the union with the dynamic-object image tree. The drawing process needs to traverse only those subtrees which have changed in the current frame or else in the immediately prior frame so that the image of previously but no longer occluded static-object faces can be redrawn. This then provides a simple means of exploiting temporal correlation (frame-to-frame coherence) in this particular setting, i.e. static view and relatively few moving objects.

Visibility computations are, of course, crucial in the evaluation of all light transport equations. The equivalence between visible surface and shadow computations was recognized at a fairly early stage. Thus, our model-modification method can be used to partition model space into regions that are homogeneous in the number of lights visible from any point in that region, which then provides a way to classify any other set to determine its light-source visibility and simultaneously detect collisions. For global illumination, a well established technique is the hemicube method [Cohen and Greenberg 85] which for each surface element projects the scene onto a half-cube whose surface has been partitioned by a grid, and visibility is approximated within each grid-cell at the midpoint. Image trees provide an alternative to this grid. For each face of the hemicube, one can use our methods to represent the image. And instead of approximating the form factors discretely, the transport can be computed exactly using contour integration [Nishita and Nakamae 85] [Campbell 91]. This then leads to a global illumination algorithm with certain similarities to that of [Campbell 91] which is also based on partitioning trees. Similarly, image trees can be used to implement light buffers [Haines and Greenberg 86], once again, with exact rather than approximate visibility, yielding a significantly simplified methodology.

Generating image trees from 3D models provides a potentially important companion to our work on a discrete-to-continuous transform in which a pixel array representation is converted into a corresponding partitioning tree representation [Rahda et al 91]. Currently, we can solve no more than the segmentation problem; texture representation remains an open issue. However, this may be enough for certain applications. Consider a robotics application in which the constituents of an external environment are known *a priori* and for which a geometric model has been constructed. The problem is to maintain a correlation between an internal geometric model and the dynamic external physical state, given an initial correlated state. One could construct two image trees, one from the discrete image provided by a camera, and the other from the current view of the geometric model. These then could be correlated by an iterative process using affine transformations, set operations (symmetric difference) and calculation of moments, and in doing so determine how the geometric model should be updated. It may also be possible to use image trees in template matching.

Examples

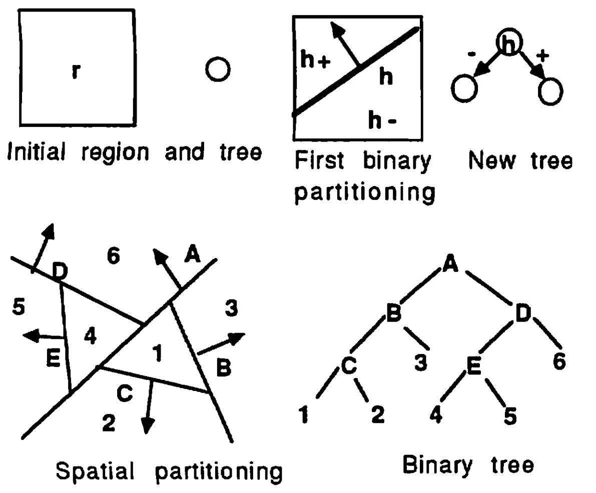
Pictures 1-5 provide a few examples of generating image trees. The number of faces for pictures 1-3 are given below for each of the three rendering methods: painter's algorithm (method 1), creating a new image tree (method 2), and modifying the model tree (method 3). For the phone handset, the difference between method 3 and the number of front-facing polygons is due to the fact that the polygons forming the sound transmitting holes in the hand set are contained in fully occluded subtrees which are condensed to a single cell (the tree has also been clipped).

object	method 1	method 2	method 3
head	705	1982	703
shuttle	499	1100	523
phone	432	353	141

In Picture 4, we have composited two 3D image trees using a union operation. Picture 5 shows a skewed view of this revealing the solid nature of the images (the full boundary of the solid images has been generated only for the purpose of illustrating their 3D nature).

Appendix

Binary space partitioning trees, also called *bsp trees* or *partitioning trees*, are defined by a generating algorithm, and for this only one operation is required: binary partitioning by a hyperplane of a region in a d-dimensional continuous space, $d > 0$. Figure A.1 illustrates this. Given a homogeneous open region **r**, a hyperplane **h** that intersects **r** is chosen using some criteria. Then **h** is used to induce a binary partitioning on **r** that generates two new d-dimensional regions, $\mathbf{r}^+ = \mathbf{r} \cap \mathbf{h}^+$ and $\mathbf{r}^- = \mathbf{r} \cap \mathbf{h}^-$, where $\mathbf{h}^+$ and $\mathbf{h}^-$ are the positive and negative *open* halfspaces of **h** respectively. Also, generated is a (d-1)-dimensional region $\mathbf{r}^0 = \mathbf{r} \cap \mathbf{h}$, called a *sub-hyperplane* (abbr. as *shp*). Thus $\mathbf{r} = \mathbf{r}^+ \cup \mathbf{r}^- \cup \mathbf{r}^0 = (\mathbf{r} \cap \mathbf{h}^+) \cup (\mathbf{r} \cap \mathbf{h}^-) \cup (\mathbf{r} \cap \mathbf{h})$. Any of these new unpartitioned homogeneous regions can be partitioned similarly, and so on recursively. When the process is terminated, the remaining unpartitioned regions, called *cells*, together with the sub-hyperplanes forms a partitioning of the initial region. (In figure A.1, the cells are labeled with numbers and the sub-hyperplanes with letters.)



Constructing a partitioning tree
Figure A.1

This process, when begun with d-space as the initial region, induces a structure on d-space in the form of a hierarchical decomposition. A partitioning tree is the computational representation of this process, and its combinatorial/syntactic form is captured by a binary tree. This tree is simply the directed graph of an asymmetric relation defined on the set of regions generated by the process where $\mathbf{r}_1 \rightarrow \mathbf{r}_2$ if $\mathbf{r}_2$ was created directly by a partitioning of $\mathbf{r}_1$. The tree also corresponds to the graph of the partial ordering of the regions induced by the subset relation. In addition, the tree can be interpreted as a type of computation graph by interpreting the arcs as intersection operations: "moving" a set s contained in a region **r** and partitioned by hyperplane **h** along a left arc from **r** to $\mathbf{r}^-$ can be interpreted as computing $s \cap \mathbf{h}^-$, and similarly for the right arc. This interpretation provides a set theoretic definition of any region $\mathbf{r}'$ as the intersection of open halfspaces corresponding to arcs on the path from the root to $\mathbf{r}'$. In figure A.1, $\textbf{cell-3} = \textbf{2-space} \cap \mathbf{A}^- \cap \mathbf{B}^+$. Consequently, if the initial region is a convex and open set, it follows that all regions of the tree are convex and open.

A partitioning tree can provide the basis of a computational object for the semantic domain of *geometric sets*. These are subsets of continuous spaces of finite dimension for which each point has an associated set of attributes (e.g. color). The partitioning tree provides an isomorphism between certain geometric entities and a combinatorial structure manipulated by algorithms; in other words, the binary tree is a syntactic entity whose intended interpretation, or model (as in Model Theory), is a geometric set. In particular, a polytope, or collection of polytopes, can be represented by associating with each cell a *membership attribute* = { **in, out** }, dividing the cells into in-cells and out-cells. The polytope may be of any topology, including multiple connected components, and have a boundary that is non-manifold and/or unbounded. All possible trees represent some topologically valid polytope, although if a tree is chosen at random, certain subtrees may correspond to the empty set or to a homogeneous region. This means that every syntactically valid tree, i.e. any binary tree with hyperplanes at internal nodes and membership attributes at leaf nodes, represents a semantically valid polytope.

For any point in d-space, its ε-neighborhood with respect to the polytope can be discovered by following the paths in the tree to any cell whose closure contains the point. This is just the standard method of inserting a point into a search tree, with the simple extension that whenever a point is found to lie on a partitioning hyperplane, both subtrees are visited. The cells reached by the traversal are exactly those lying in the point's ε-neighborhood [Thibault and Naylor 87] .

Any central projection using linear projectors (rays) determines a partial ordering, called a *visibility priority ordering*, on the regions of any partitioning tree. This ordering depends only upon the center of projection. The total priority ordering induced by any single ray on the subset of the regions it intersects is consistent with this "global" partial ordering. The ordering is possible because for a ray **t** and hyperplane **h**, their intersection is a single point, unless **t** lies in **h**. This intersection point partitions **t** into near, on and far subsets. This implies that any set in the near-halfspace of **h** has priority over any sets lying in **h** which in turn has priority of any sets in the far-halfspace. Given a partitioning tree representation of polyhedra, discovering that the viewing position is in say the positive halfspace of a partitioning hyperplane **h** at node **v** means that all sets represented by the positive subtree of **v** have priority over any sets lying in **h** which then have priority over those sets represented by the negative subtree. One can apply this local ordering recursively to generate a total priority ordering of all sets represented by the tree. (See [Schumacker et al 69] or [Sutherland, Sproull and Schumacker 74], and [Fuchs, Kedem and Naylor 80] or [Naylor 81]).

Along similar lines, efficient ray-tracing algorithms have been devised [Naylor and Thibault 86] which exploit both the convex decomposition and the inherent hierarchical search structure. Calculation of shadows due to point light sources is addressed in [Chin and Feiner 89] and due to area light sources in [Chin and Feiner 92] and [Campbell 91]. Use of partitioning trees for global illumination calculations can be found in [Fussell and Campbell 90] [Campbell 91]. Algorithms for set operations are presented in [Thibault and Naylor 87] and [Naylor, Amanatides and Thibault 90].

In [Naylor 81], it was shown that partitioning trees could represent arrangements of hyperplanes, and the complexity of the arrangements was used to give a bound of $\Theta(\mathbf{n^d})$ on the size of the largest possible partitioning tree formed using **n** hyperplanes in d-space. It was also shown that a set of disjoint **(d-1)-faces** could result in a tree of size $\Omega(\mathbf{n^{d-1}})$. In [Paterson and Yao 90] algorithms are given for converting a set of non-intersecting faces to a partitioning tree of size $\Theta(\mathbf{n^{d-1}})$ in time $O(\mathbf{n^{d+1}})$, $\mathbf{d} > 3$, which is reduced to $\Theta(\mathbf{n^2})$ and $O(\mathbf{n^3})$ for 3D. In 2D, the tree size and run time are both $O(\mathbf{n \log n})$. A convex n-gon can be represented by a tree of size $\Theta(\mathbf{n})$ and depth $\Theta(\mathbf{\log n})$ and two such trees can be merged in $O(\mathbf{n \log n})$ [Naylor 92]. Two arbitrary trees each of size **n** can be merged in $\Theta(\mathbf{n^d})$ for $\mathbf{d} = 2, 3$ or 4 [Naylor, Thibault and Amanatides 90]. However, empirical results in [Naylor 81] and much subsequent experience indicate that polygonal models of real objects result in trees much closer to $O(\mathbf{n \log n})$.

Partitioning trees are the same computational structure as *linear decision trees* [Rabin 72], which have been used to prove lower bounds on various problem, e.g. that sorting is $\Omega(\mathbf{n \log n})$. Another application of this structure, concerned primarily with representing a finite set of points is called *polygon trees* [Willard 82] or *partition trees.*

References

[Abram, Westover and Whitted 85]
Greg Abram, Lee Westover and Turner Whitted, "Efficient Alias-free Rendering using Bit-masks and Look-up Tables", **Computer Graphics** Vol. 19(3), pp. 53-59, (July 1985).

[Carpenter 84]
Loren Carpenter, "The A-buffer, an Antialiased Hidden Surface Method", **Computer Graphics** Vol. 18(3), pp. 103-108 (July 1984).

[Catmull 78]
Edwin Catmull, "A Hidden Surface Algorithm with Anti-Aliasing", **Computer Graphics** Vol. 12(3), pp. 6-10 (July 1978).

[Campbell and Fussell 90]
A.T. Campbell and Don Fussell, "Adaptive Mesh Generation for Global Diffuse Illumination", **Computer Graphics** Vol. 24(4), pp. 155-164, (August 1990).

[Campbell 91]
A.T. Campbell "Modeling Global Diffuse for Image Synthesis", Ph.D. Dissertation, Department of Computer Science, University of Texas at Austin, (1991).

[Chin and Feiner 89]
Norman Chin and Steve Feiner, "Near Real-Time Shadow Generation Using BSP Trees", **Computer Graphics** Vol. 23(3), pp. 99-106, (July 1989).

[Chin and Feiner 92]
Norman Chin and Steve Feiner, "Fast Object-Precision Shadow Generation for Area Light Sources Using BSP Trees", **Symp. on 3D Interactive Graphics**, (March 1992).

[Cohen and Greenberg 85]
Michael F. Cohen and Donald P. Greenberg, "The Hemi-Cube: A Radiosity Solution for Complex Environments", **Computer Graphics** Vol. 19(3), pp. 31-40, (July 1985).

[Duff 85]
Tom Duff, "Compositing 3-D Rendered Images", **Computer Graphics** Vol. 19(3), pp. 41-44, (July 1985).

[Fiume and Fournier 83]
Eugene Fiume and Alain Fournier, "A Parallel Scan Conversion Algorithm with Anti-Aliasing for a General-Purpose Ultracomputer", **Computer Graphics** Vol. 17(3), pp. 141-150, (July 1983).

[Fuchs, Kedem, and Naylor 80]
H. Fuchs, Z. Kedem, and B. Naylor, "On Visible Surface Generation by a Priori Tree Structures," **Computer Graphics** Vol. 14(3), pp. 124-133, (June 1980).

[Harp 86]
Keith Harp, "An Empirical Study of Visible Surface Algorithms" , Masters Thesis, School of Information and Computer Science, Georgia Institute of Technology, (Sept. 1986).

[Haines and Greenberg 86]
Eric A. Haines and Donald P. Greenberg, "The Light Buffer: a Shadow-Testing Accelerator", **IEEE Computer Graphics and Applications** Vol. 6(9), pp. 6-16, (Sept. 1986).

[Mitchell 87]
Don P. Mitchell, "Generating Antialiased Images at Low Sampling Densities" **Computer Graphics**, Vol. 21(4), pp. 65-72, (July 1987).

[Mitchell and Netravali 88]
Don P. Mitchell and Arun N. Netravali, "Reconstruction Filters in Computer Graphics" **Computer Graphics**, Vol. 22(4), pp. 221-228, (August 1988).

[Naylor 81]
Bruce F. Naylor, "A Priori Based Techniques for Determining Visibility Priority for 3-D Scenes," Ph.D. Thesis, University of Texas at Dallas (May 1981).

[Naylor and Thibault 86]
Bruce F. Naylor and William C. Thibault, "Application of BSP Trees to Ray-Tracing and CSG Evaluation", Technical Report GIT-ICS 86/03, School of Information and Computer Science, Georgia Institute of Technology, Atlanta, Georgia 30332 (February 1986).

[Naylor 90a]
Bruce F. Naylor, "Binary Space Partitioning Trees as an Alternative Representation of Polytopes," **Computer Aided Design**, Vol. 22(4), (May 1990).

[Naylor 90b]
Bruce F. Naylor, "SCULPT: an Interactive Solid Modeling Tool," Proceeding of *Graphics Interface* (May 1990).

[Naylor, Amanatides and Thibault 90]
Bruce F. Naylor, John Amanatides and William C. Thibault, "Merging BSP Trees Yields Polyhedral Set Operations", **Computer Graphics** Vol. 24(4), pp. 115-124, (August 1990).

[Naylor 92]
Bruce F. Naylor, "Constructing Good Partitioning Trees," manuscript in preparation.

[Nishita and Nakamae 85]
Tomoyuki Nishita and Eihachiro Nakamae, "Continuous Tone Representation of Three-Dimensional Objects Taking Account of Shadows and Interreflection", **Computer Graphics** Vol. 19(3), pp. 23-30, (July 1985).

[Porter and Duff 84]
Thomas Porter and Tom Duff, "Compositing Digital Images", **Computer Graphics** Vol. 18(3), pp. 253-259, (July 1984).

[Rabin 72]
Michael O. Rabin, "Proving Simultaneous Positivity of Linear Forms", **Journal of Computer and Systems Science**, Vol. 6, pp. 639-650 (1972).

[Rahda et al 91]
Hayder Rahda, Riccardo Leonardi, Martin Vetterli and Bruce Naylor, "Binary Space Partitioning Tree Representation of Images", **Visual Communications and Image Representation**, Vol. 2(3), pp. 201-221, (Sept. 1991).

[Schumacker et al 69]
R. A. Schumacker, R. Brand, M. Gilliland, and W. Sharp, "Study for Applying Computer-Generated Images to Visual Simulation," AFHRL-TR-69-14, U.S. Air Force Human Resources Laboratory (1969).

[Sharir and Overmars 92]
Micha Sharir and Mark H. Overmars, "A Simple Output-Sensitive Algorithm for Hidden Surface Removal," **ACM Transactions on Graphics** Vol. 11(1), (1992).

[Sutherland, Sproull and Schumacker 74]
I.E. Sutherland, R.F. Sproull and R. A. Schumacker, "A Characterization of Ten hidden Surface Algorithms," **ACM Computing Surveys** Vol. 6(1), (1974).

[Teller and Sequin 92]
Seth J. Teller and Carlo H. Sequin, "Visibility Preprocessing For Interactive Walkthroughs", **Computer Graphics** Vol. 25(4), pp. 61-69, (July 1991).

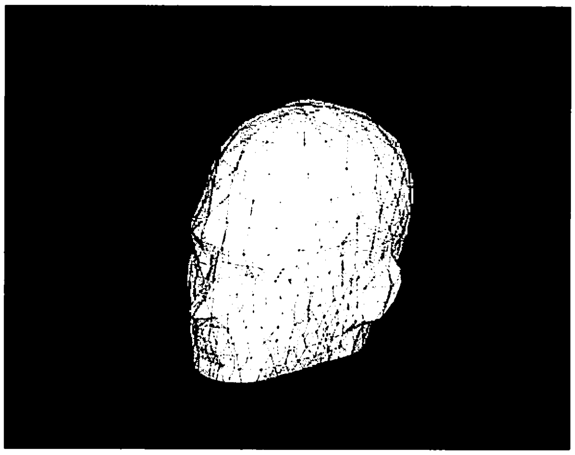
[Thibault and Naylor 87]
W. Thibault and B. Naylor, "Set Operations On Polyhedra Using Binary Space Partitioning Trees," **Computer Graphics** Vol. 21(4), pp. 153-162, (July 1987).

[Warnock 69]
John E. Warnock, "A Hidden-Surface Algorithm for Computer Generated Halftone Pictures", Computer Science Department, University of Utah, TR 4-15, (June 1969).

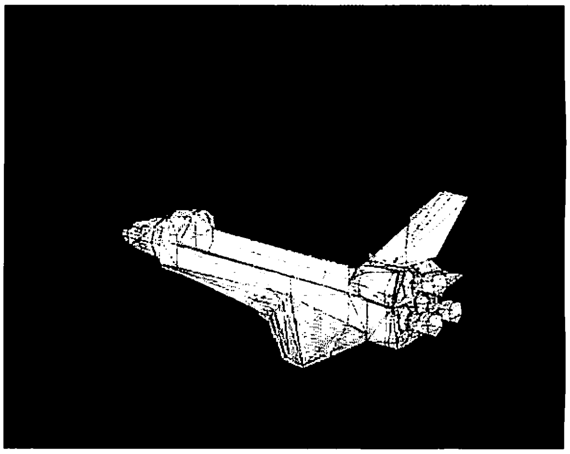
[Weiler 80]
Kevin Weiler, "Polygon Comparison Using a Graph Representation", **Computer Graphics** Vol. 14(3), pp. 10-18 (July 1980).

[Weiler and Atherton 77]
Kevin Weiler and P. Atherton, "Hidden Surface Removal Using Polygon Area Sorting", **Computer Graphics** Vol. 11(3), pp. 103-108 (July 1984).

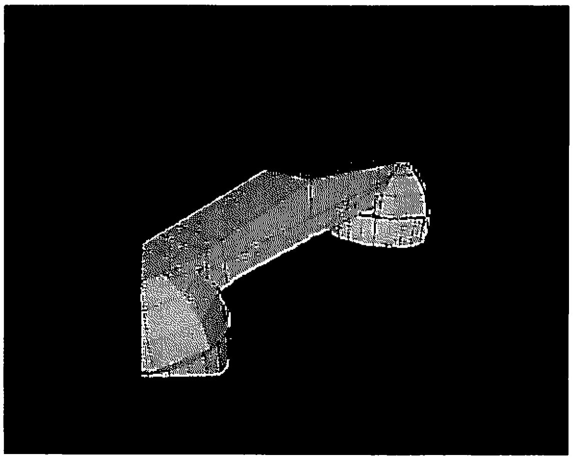
[Willard 82]
Dan E. Willard, "Polygon Retrieval", **SIAM Journal of Computing**, Vol. 11(1), pp. 149-165 (Feb. 1982).



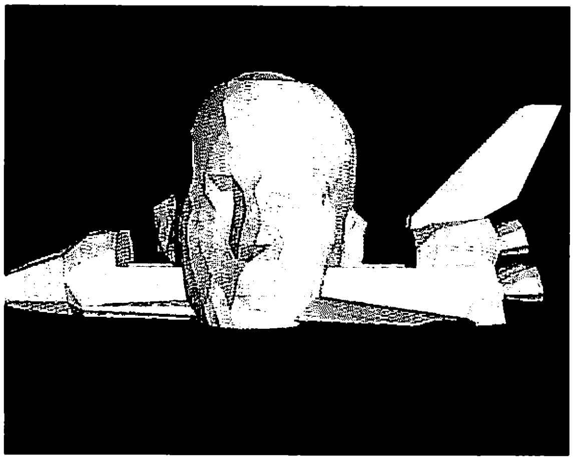
1



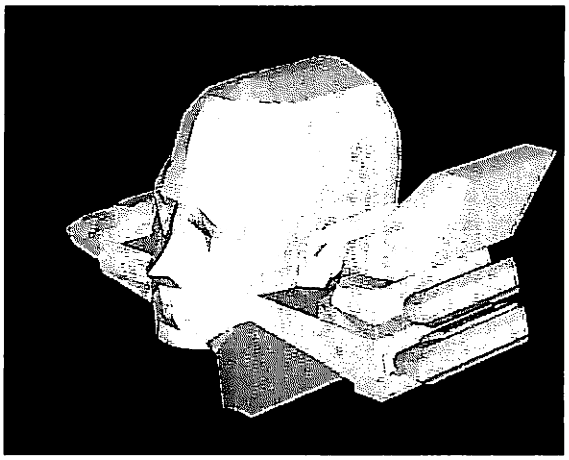
2



3



4



5

Escape-time Visualization Method for Language-restricted Iterated Function Systems

Przemyslaw Prusinkiewicz
Mark S. Hammel
Department of Computer Science
University of Calgary
Calgary, Alberta, Canada T2N 1N4

Abstract

The escape-time method was introduced to generate images of Julia and Mandelbrot sets, then applied to visualize attractors of iterated function systems. This paper extends it further to language-restricted iterated function systems (LRIFS's). They generalize the original definition of IFS's by providing means for restricting the sequences of applicable transformations. The resulting attractors include sets that cannot be generated using ordinary IFS's. The concepts of this paper are expressed using the terminology of formal languages and finite automata.

Keywords: fractal, iterated function system, escape-time method, graphics algorithm, formal language, finite automaton.

1. Introduction

Although mathematicians have explored the properties of fractals since the turn of century, they could not visualize the objects of their study without the aid of computers. Computer graphics made it possible to recognize the beauty of fractals, and turned them into an art form [13]. Peitgen and Richter [14] perfected and popularized images of Julia and Mandelbrot sets. Many of them were created using the escape-time method. In its original setting, it consisted of testing how fast points z outside the attractor diverged to infinity while iterating function $z \longrightarrow z^2 + c$ in the complex plane. The resulting values were interpreted as colors in a two-dimensional image, or height values in a "fractal landscape" [15, Section 2.7].

The escape-time visualization method was extended from Julia sets to iterated function systems [12] by Barnsley [2] and Prusinkiewicz and Sandness [18]. This paper extends it further to language-restricted iterated function systems, introduced in [16]. They generalize the original definition of IFS's by providing means for restricting the sequences of applicable transformations to a particular set. The resulting attractors form a larger class than those generated using ordinary IFS's. The definition of an LRIFS leaves open the mechanism for sequencing transformations, thus LRIFS's incorporate the earlier generalizations committed to a particular mechanism, such as hierarchical IFS's [4], sofic systems [1], recurrent IFS's [3], Markov IFS's [21], mixed IFS's [5], controlled IFS's [17], and mutually recursive function systems [7, 8]. Several other authors considered similar generalizations without giving them a name. Visualization of the attractors of generalized IFS's has been addressed by Hart [9], referring to his earlier results with DeFanti [10].

This paper is organized as follows. Sections 2 and 3 summarize the background material related to formal languages and iterated function systems. Section 4 presents the escape-time method for IFS's in a way suitable for further extensions. Section 5 defines the language-restricted iterated function systems. The escape-time method is extended to LRIFS's in Section 6. A special case of regular languages is considered and illustrated using examples in Section 7. Section 8 summarizes the results.

2. Formal languages

An *alphabet* V is as a finite nonempty set of *symbols* or *letters*. A *string* or *word* over alphabet V is a finite sequence of zero or more letters of V, whereby the same letter may occur several times. The total number of letters in a word w is called its *length*, and denoted length(w). The word of zero length is called the *empty word* and denoted ϵ. The *concatenation* of words $x = a_1 a_2 \dots a_m$

and $y = b_1b_2\dots b_n$ is the word formed by extending the sequence of symbols x with the sequence y, thus $xy = a_1a_2\dots a_mb_1b_2\dots b_n$. If $xy = w$ then the word x is called the *prefix* of w, denoted $x \prec w$. The remaining word y is called the *suffix*. We assume that the relation $\prec$ is reflexive, that is, $w \prec w$. The n-fold concatenation of a word w with itself is called its n-th *power*, and denoted w^n. By definition, $w^0 = \epsilon$ for any w. If $w = a_1a_2\dots a_n$, then the word $w^R = a_n\dots a_2a_1$ is called the *mirror image* of w. It can be shown that $(xy)^R = y^Rx^R$ for any words x and y.

The set of all words over V is denoted by $V^\star$, and the set of nonempty words by V^+. A *formal language* over an alphabet V is a set L of words over V, hence $L \subset V^\star$. The concatenation and mirror image of words are extended to languages as follows:

$$\begin{aligned} L_1L_2 &= \{xy : x \in L_1 \,\&\, y \in L_2\}, \\ L^R &= \{w^R : w \in L\}. \end{aligned}$$

A language L is *prefix extensible* if there exists a word $v \in V^+$ such that $vL \subset L$. In other words, $vw \in L$ for every word $w \in L$. The *right derivative* of a language $L \subset V^\star$ with respect to a word $v \in V^\star$ is the language:

$$L//v = \{w \in V^\star : vw \in L\}.$$

The set of all prefixes of a language L is called the *prefix closure* of L:

$$\mathcal{P}(L) = \{x \in V^\star : (\exists w \in L)\, x \prec w\}.$$

3. Iterated function systems

Let $\langle X, d\rangle$ be a complete metric space with support X and distance function d (in this paper, we will only consider the plane with the Euclidean distance). A function $F : X \longrightarrow X$ is called a *contraction* in X if there is a constant $r < 1$ such that

$$d(F(P), F(Q)) \le rd(P,Q)$$

for all $P, Q \in X$. The parameter r is called the *Lipschitz constant* of F.

An *iterated function system* (IFS) in X is a quadruplet $\mathcal{I} = \langle X, \mathcal{F}, V, h, \rangle$, where:

- X is the underlying metric space,
- $\mathcal{F}$ is a set of contractions in X,
- V is an alphabet of contraction labels,
- $h : V \longrightarrow \mathcal{F}$ is a *labeling function*, taking the letters of alphabet V to the contractions from $\mathcal{F}$.

In the literature, an IFS is usually defined as the pair $\langle X, \mathcal{F}\rangle$. We extend this definition by specifying the alphabet V and the labeling function h to facilitate the introduction of language-restricted IFS's in Section 5.

The function h is extended to words and languages over V using the equations:

$$\begin{aligned} h(a_1a_2\dots a_n) &= h(a_1)\circ h(a_2)\circ\dots\circ h(a_n), \\ h(L) &= \bigcup_{w\in L} h(w), \end{aligned}$$

where the symbol $\circ$ denotes function composition,

$$x \circ f_1 \circ f_2 \circ \dots \circ f_n = f_n(\cdots(f_2(f_1(x)))\cdots).$$

The *attractor* of an IFS $\mathcal{I}$ is the smallest nonempty set $\mathcal{A} \subset X$, closed with respect to all transformations of $\mathcal{F}$, and closed in the set-theoretic sense. Hutchinson showed that the attractor of an arbitrary IFS always exists and is unique [12]. Consequently, it can be found by selecting a point $P \in \mathcal{A}$, and applying to it all possible sequences of transformations from $\mathcal{F}$:

$$\mathcal{A} = \mathrm{cl}(P \circ h(V^\star)),$$

where the symbol cl represents the set-theoretic closure of the argument set. There are several methods for finding the initial point $P \in \mathcal{A}$. For example, the fixed point of any transformation $F \in \mathcal{F}$ is known to belong to $\mathcal{A}$ [12].

A legible notation for specifying transformations is needed while defining particular IFS's. In this paper we express transformations by composing operations of translation, rotation, and scaling in an underlying Cartesian coordinate system. The following symbols are used:

- $t(a, b)$ is a translation by vector (a, b).
- $a(\alpha)$ is a rotation by (oriented) angle α with respect to the origin of the coordinate system. The angles are expressed in degrees.
- $s(r_x, r_y)$ is a scaling with respect to the origin of the coordinate system: $x' = r_xx$ and $y' = r_yy$. If $r_x = r_y = r$, we write $s(r)$ instead of $s(r, r)$.

For example, Figure 1 shows the attractor of an IFS $\mathcal{I} = \langle X, \mathcal{F}, V, h\rangle$, where the set $\mathcal{F}$ consists of two transformations:

$$\begin{aligned} F_1 &= s\left(\frac{\sqrt{2}}{2}\right) \circ r(45), \\ F_2 &= s\left(\frac{\sqrt{2}}{2}\right) \circ r(135) \circ t(0,1). \end{aligned}$$

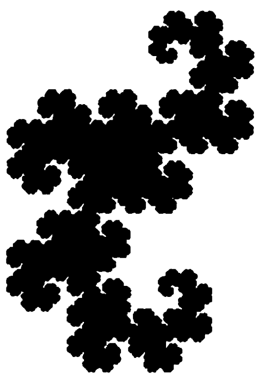


Figure 1: The dragon curve

4. The escape-time method

Consider an IFS $\mathcal{I} = \langle X, \mathcal{F}, V, h \rangle$, where all functions $F \in \mathcal{F}$ are invertible. Let $\tilde{h}(a)$ denote the inverse of the contraction $F = h(a) \in \mathcal{F}$, or $\tilde{h}(a) = (h(a))^{-1}$. The function $\tilde{h}$ is extended to words and languages over V in a way similar to h:

$$\begin{aligned} \tilde{h}(a_1 a_2 \ldots a_n) &= \tilde{h}(a_1) \circ \tilde{h}(a_2) \circ \ldots \circ \tilde{h}(a_n), \\ \tilde{h}(L) &= \bigcup_{w \in L} \tilde{h}(w). \end{aligned}$$

A *trajectory* of a point Q with respect to a word $w \in V^*$ is the set:

$$\mathrm{Tr}(Q, w) = \{Q \circ \tilde{h}(x) : x \prec w\}.$$

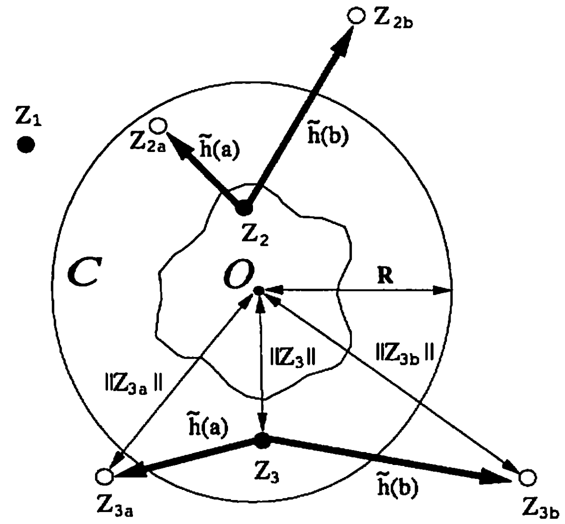
The length of w is referred to as the length of the trajectory. The escape-time method for visualizing the attractor of $\mathcal{I}$ is based on the following Theorem, proven in [18]:

Theorem 1. (a) If a starting point Q belongs to the attractor $\mathcal{A}$ of an IFS $\mathcal{I}$, there exists an infinitely long trajectory entirely included in $\mathcal{A}$. (b) If the point Q does not belong to $\mathcal{A}$, all trajectories diverge to infinity.

To estimate the speed with which the divergence occurs, we enclose the attractor in a circle. Since attractors of IFS's are bounded, it is always possible to find a circle $\mathcal{C}$ of a finite radius R, completely enclosing $\mathcal{A}$. The escape time of a point $Q \notin \mathcal{A}$ is then defined as the length of the longest trajectory included in $\mathcal{C}$:

$$E_1(Q) = \max_{w \in V^*} \{\mathrm{length}(w) : \mathrm{Tr}(Q, w) \subset \mathcal{C}\}.$$

According to this definition, the function $E_1(Q)$ is integer-valued. In order to represent the escape time



- $\mathrm{res}(Z_1, a) = \mathrm{res}(Z_1, b) = 0$, since $Z_1 \notin \mathcal{C}$,
- $\mathrm{res}(Z_2, a) = 0$, since $Z_2 \circ \tilde{h}(a) = Z_{2a} \in \mathcal{C}$,
- $\mathrm{res}(Z_3, a) > \mathrm{res}(Z_3, b)$, since $\|Z_{3a}\| < \|Z_{3b}\|$.

Figure 2: Illustration of the residual terms $\mathrm{res}(Z, a)$.

with a higher precision, Hepting *et al.* [11] introduced a residual term that reflects the distance between the last point in the escape trajectory $Tr(Q, w)$ and the border of circle $\mathcal{C}$:

$$E_2(Q) = \max_{wa \in V^*} \{\mathrm{length}(w) + \mathrm{res}(Q \circ \tilde{h}(w), a) : \mathrm{Tr}(Q, w) \subset \mathcal{C}\}.$$

Let $Z = Q \circ \tilde{h}(w)$. The function res : $X \times V \to [0, 1)$ is defined as follows:

$$\mathrm{res}(Z, a) = \begin{cases} \dfrac{\log R - \log \|Z\|}{\log \|Z \circ \tilde{h}(a)\| - \log \|Z\|} & \text{if } Z \in \mathcal{C} \text{ and } Z \circ \tilde{h}(a) \notin \mathcal{C}, \\ 0 \quad \text{otherwise.} \end{cases} \tag{1}$$

The norm symbol $\|Z\|$ denotes the distance between point Z and the center O of circle $\mathcal{C}$, thus $\|Z\| = d(Z, O)$. The function $\mathrm{res}(Z, a)$ has the following properties (Figure 2):

- it takes a nonzero value if point Z lies inside the circle $\mathcal{C}$ and its image $Z \circ \tilde{h}(a)$ lies outside this circle;
- it tends to 0 if the point Z approaches the boundary of circle $\mathcal{C}$, and to 1 if the image $Z \circ \tilde{h}(a)$ approaches this boundary.

Observe that, when the length of the longest trajectory included in $\mathcal{C}$ is incremented as a result of moving the starting point Q towards the attractor, the largest residual term changes its value from 1 to 0. Consequently, $E_2(Q)$ is a continuous function of the position of point Q in the domain $X\backslash\mathcal{A}$. For a formal proof of this property see [11]. A justification of the choice of Formula 1 is given in the Appendix.

The escape-time functions $E_1(Q)$ and $E_2(Q)$ are not defined inside the attractor $\mathcal{A}$, as one can find there infinite sequences of points remaining in $\mathcal{A}$ and therefore remaining in the circle $\mathcal{C}$. In order to make the definition of the escape time computationally effective, we evaluate the escape trajectories up to a predefined maximum length m. The escape-time functions, limited in this way, can be computed in the entire space X using the following formulae:

$$\underline{E}_1(Q,m) =$$
$$\begin{cases} 0 & \text{if } Q \notin \mathcal{C} \text{ or } m = 0, \\ 1 + \max_{a \in V}\{\underline{E}_1(Q \circ \tilde{h}(a), m-1)\} & \text{otherwise.} \end{cases}$$

$$\underline{E}_2(Q,m) =$$
$$\begin{cases} 0 & \text{if } Q \notin \mathcal{C} \text{ or } m = 0, \\ \max_{a \in V}\{res(Q,a)\} & \begin{array}{l}\text{if } Q \in \mathcal{C},\ m > 0, \text{and} \\ Q \circ \tilde{h}(a) \notin \mathcal{C} \text{ for all } a \in V,\end{array} \\ 1 + \max_{a \in V}\{\underline{E}_2(Q \circ \tilde{h}(a), m-1)\} & \text{otherwise.} \end{cases}$$

It is intuitively clear that $\underline{E}_1(Q,m) = E_1$ for all points Q with the escape time $E_1(Q)$ less than m, since the recursive formula evaluates step-by-step the same trajectories as its non-recursive counterpart. Similarly, $\underline{E}_2(Q,m) = E_2(Q)$ for all points Q such that $E_2(Q) < m$. Rigorous proofs of these equalities can be carried out by induction on m.

Figure 3 visualizes the dragon curve from Figure 1 using the continuous escape-time function $\underline{E}_2(Q,m)$. The inverse functions are:

$$\begin{aligned} F_1^{-1} &= r(-45) \circ s(\sqrt{2}), \\ F_2^{-1} &= t(0,-1) \circ r(-135) \circ s(\sqrt{2}). \end{aligned}$$

It is assumed that the circle $\mathcal{C}$ has radius R equal to 5, and the limit m is equal to 20. The values of function $\underline{E}_2(Q,m)$ are interpreted as a height field.

5. Language-restricted IFS's

A *language-restricted* iterated function system (LRIFS) is a quintuplet $\mathcal{I}_L = \langle X, \mathcal{F}, V, h, L\rangle$, where $X, \mathcal{F}, V$, and

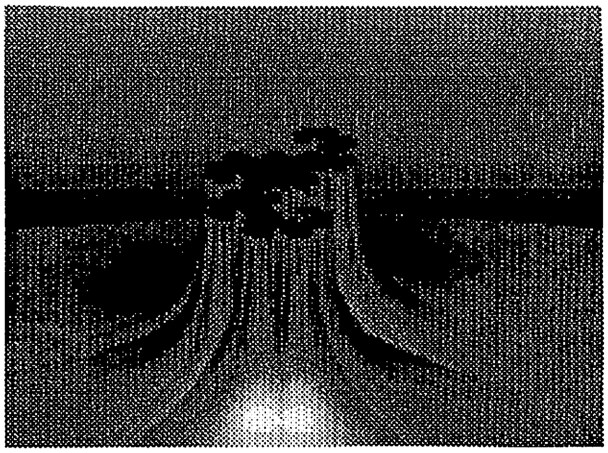


Figure 3: The dragon curve visualized using the escape-time method

h form an "ordinary" IFS, and $L \subset V^*$ is a language over the alphabet V.

Consider a starting point P that belongs to the attractor $\mathcal{A}$ of the IFS $\mathcal{I}$, and let $\mathcal{A}_L(P)$ denote the closure of the image of P with respect to the transformations $h(L)$. The following inclusion holds:

$$\mathcal{A}_L(P) = \text{cl}(P \circ h(L)) \subset \text{cl}(P \circ h(V^*)) = \mathcal{A}.$$

Thus, the set $\mathcal{A}_L(P)$ generated by the LRIFS $\mathcal{I}_L$ with the starting point $P \in \mathcal{A}$ is a subset of the attractor $\mathcal{A}$. For example, consider an LRIFS $\mathcal{F}_L = \langle X, \mathcal{F}, V, h, L\rangle$, where:

- the space X is the plane,
- the IFS $\mathcal{F}$ consists of four transformations:

$$\begin{aligned} F_1 &= s(0.5), \\ F_2 &= s(0.5) \circ t(0,0.5), \\ F_3 &= s(0.5) \circ r(45) \circ t(0,1), \\ F_4 &= s(0.5) \circ r(-45) \circ t(0,1), \end{aligned}$$

- the alphabet V consists of four letters a, b, c, d,
- the homomorphism h is defined by:

$$h(a) = F_1,\ \ h(b) = F_2,\ \ h(c) = F_3,\ \ h(d) = F_4,$$

- the language L consists of words in which no letter c or d is followed by an a or b. [1]

[1]Thus, L is defined by the regular expression:

$$L = (a \cup b)^*(c \cup d)^*.$$

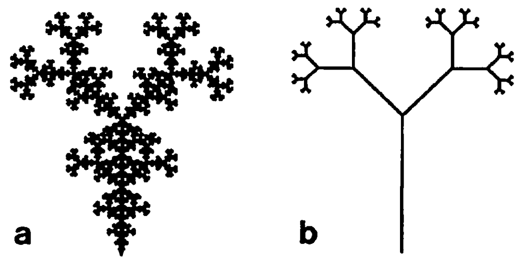


Figure 4: Attractor $\mathcal{A}$ and its subset $\mathcal{A}_L(P)$

Figure 4 compares the attractor $\mathcal{A}$ of the IFS $\mathcal{I}$ with the set $\mathcal{A}_L(P)$ generated by the LRIFS $\mathcal{I}_L$ using the starting point $P = (0,0)$. Clearly, the branching structure of Figure (b) is a subset of the original attractor (a).

In general, the set $\mathcal{A}_L(P)$ depends on the choice of the starting point P. Nevertheless, if the language L is prefix extensible (i.e., $vL \subset L$), the smallest set $\mathcal{A}_L$ does exist and can be found as $\mathrm{cl}(P_0 \circ h(L))$, where P_0 is the invariant point of the transformation $h(v)$. This results from the following inclusions, satisfied for any $P \in X$:

$$\begin{aligned}\mathrm{cl}(P_0 \circ h(L)) &= \mathrm{cl}((\lim_{n\to\infty} P \circ h(v^n)) \circ h(L)) \\ &= \lim_{n\to\infty} \mathrm{cl}(P \circ h(v^n L)) \subset \mathrm{cl}(P \circ h(L)).\end{aligned}$$

The limits are calculated in the space of all closed nonempty bounded subsets of the space X with the Hausdorff metric [16]. By analogy with the "ordinary" IFS's, we call $\mathcal{A}_L$ the attractor of the LRIFS $\mathcal{I}_L$.

6. The escape-time method for LRIFS's

While extending the escape-time method to LRIFS's, we consider mirror images of words and use the following lemma.

Lemma. Consider IFS $\mathcal{I} = \langle X, \mathcal{F}, V, h\rangle$, and let all functions $F = h(a) \in \mathcal{F}$ be invertible. Then for any word $w \in V^*$, the equality $\tilde{h}(w) = (h(w^R))^{-1}$ holds.

Proof. The set $\mathcal{F}$ forms a group of transformations with the operations of function composition and inversion, thus $(F_i \circ F_j)^{-1} = F_j^{-1} \circ F_i^{-1}$ for any $F_i, F_j \in \mathcal{F}$. Consequently, the following equalities are true for any word $w = a_1a_2\ldots a_n \in V^*$:

$$\begin{aligned}\tilde{h}(w) &= \tilde{h}(a_1a_2\ldots a_n) \\ &= \tilde{h}(a_1)\circ\tilde{h}(a_2)\circ\ldots\circ\tilde{h}(a_n) \\ &= (h(a_1))^{-1}\circ(h(a_2))^{-1}\circ\ldots\circ(h(a_n))^{-1} \\ &= (h(a_n)\circ\ldots\circ h(a_2)\circ h(a_1))^{-1} \\ &= (h(a_n\ldots a_2a_1))^{-1} = (h(w^R))^{-1}. \ \square\end{aligned}$$

The escape-time method for LRIFS's is based on the following extension of Theorem 1 from Section 4:

Theorem 2. Consider an LRIFS $\mathcal{I}_L = \langle X, \mathcal{F}, V, h, L\rangle$, and assume that the language L is prefix extensible, $vL \subset L$. Denote by $\mathcal{A}$ the attractor of the IFS $\langle X, \mathcal{F}, V, h\rangle$, and by $\mathcal{A}_L$ the attractor of $\mathcal{I}_L$. (a) If a starting point $Q \in X$ belongs to the attractor $\mathcal{A}_L$, then for any $n \geq 0$ there exists a word w in the prefix closure $\mathcal{P}(L^R)$ such that $\mathrm{length}(w) \geq n$ and $\mathrm{Tr}(Q,w) \subset \mathcal{A}$. (b) If the point Q does not belong to $\mathcal{A}_L$, all trajectories $\mathrm{Tr}(Q,w)$ with $w \in \mathcal{P}(L^R)$ diverge to infinity as $\mathrm{length}(w) \to \infty$.

Proof. (a) Let P_0 denote the invariant point of the transformation $h(v)$. According to the definition of the attractor $\mathcal{A}_L$, there exists a word $y \in L$ such that $P_0 \circ h(y) = Q$.[2] Since $P_0 = P_0 \circ h(v)$, the equality $P_0 \circ h(v^i y) = Q$ holds for any $i \geq 0$. Let i satisfy the inequality $\mathrm{length}(v^i y) \geq n$, and $w = (v^i y)^R$. The word w belongs to L^R and henceforth to $\mathcal{P}(L^R)$, has length greater than or equal to n, and maps point Q to the point $P_0 \in \mathcal{A}_L$:

$$Q \circ \tilde{h}(w) = Q \circ (h(v^i y))^{-1} = P_0 \in \mathcal{A}_L.$$

In order to show that the entire trajectory $\mathrm{Tr}(Q,w)$ is included in the attractor $\mathcal{A}$, let us consider an arbitrary partition of the word w into a prefix x_1 and a suffix x_2; thus $x_1x_2 = w$. From the equality

$$Q \circ \tilde{h}(w) = Q \circ \tilde{h}(x_1) \circ \tilde{h}(x_2) = P_0$$

it follows that

$$\begin{aligned}Q \circ \tilde{h}(x_1) &= P_0 \circ (\tilde{h}(x_2))^{-1} \\ &= P_0 \circ h(x_2^R) \in P_0 \circ h(V^*) \subset \mathcal{A}.\end{aligned}$$

Since this argument holds for any $x \prec w$, we obtain:

$$\mathrm{Tr}(Q,w) = \{Q \circ \tilde{h}(x) : x \prec w\} \subset \mathcal{A}.$$

(b) Let C be an arbitrary circle enclosing the attractor $\mathcal{A}$, and R denote the radius of C. We have to prove that if $Q \notin \mathcal{A}_L$, there exists a number $n \geq 0$ such that for any word $w \in \mathcal{P}(L^R)$ of length greater then or equal to n, the escape trajectory $\mathrm{Tr}(Q,w)$ is not entirely included in C. Let D denote the distance between point Q and the attractor $\mathcal{A}_L$, and r_{max} be the largest Lipschitz constant found among the transformations $F \in \mathcal{F}$. Since $D > 0$ and $r_{max} < 1$, there exists a number $n \geq 0$ such that $2Rr_{max}^n < D$. Consider an arbitrary word $w \in \mathcal{P}(L^R)$ with $\mathrm{length}(w) \geq n$, and let $wy \in L^R$, or

[2]Strictly speaking, there exists a word $y \in L$ such that $P_0 \circ h(y)$ is arbitrarily close to Q.

$y^R w^R \in L$. Then there exist points $P_0, P \in \mathcal{A}_L$ such that $P_0 \circ h(y^R w^R) = P$. We decompose the last equality by introducing an intermediate point P':

$$P_0 \circ h(y^R) = P' \text{ and } P' \circ h(w^R) = P.$$

It follows that

$$P \circ \tilde{h}(w) = P' = P_0 \circ h(y^R) \in P_0 \circ h(V^*) \subset \mathcal{A}.$$

The distance between points P and Q is at least D, and the Lipschitz constant of the composite transformation $\tilde{h}(w) = (h(w^R))^{-1}$ is at least r_{max}^{-n}, thus

$$\begin{aligned} d(Q \circ \tilde{h}(w), P \circ \tilde{h}(w)) &\geq d(Q, P) r_{max}^{-n} \\ &\geq D r_{max}^{-n} > 2R. \end{aligned}$$

Since $P \circ \tilde{h}(w) = P' \in \mathcal{A} \subset \mathcal{C}$, and the distance of $Q \circ \tilde{h}(w)$ from P' is greater than the diameter of $\mathcal{C}$, the point $Q \circ \tilde{h}(w)$ must lie outside of $\mathcal{C}$, or

$$\mathrm{Tr}(Q, w) \not\subset \mathcal{C}. \quad \square$$

Theorem 2 reveals an analogy between the escape trajectories of an LRIFS and an ordinary IFS. In both cases we find infinitely long trajectories confined to $\mathcal{A}$ if the starting point Q belongs to the attractor — respectively $\mathcal{A}_L$ or $\mathcal{A}$. For a point Q outside an attractor, all trajectories diverge to infinity as their length increases. However, in the case of an ordinary IFS we consider escape trajectories with respect to all possible words $w \in V^*$, while in the case of an LRIFS the words w are confined to the prefix closure $K = \mathcal{P}(L^R)$.

As a result of these observations, we can extend the escape-time formulae from Section 4 to LRIFS's as follows:

$$E_{L1}(Q) = \max_{w \in K}\{\text{length}(w) : \mathrm{Tr}(Q, w) \subset \mathcal{C}\},$$

$$E_{L2}(Q) = \max_{wa \in K}\{\text{length}(w) + \text{res}(Q \circ \tilde{h}(w), a) : \mathrm{Tr}(Q, w) \subset \mathcal{C}\}.$$

In the recursive counterparts of these functions, the key issue is the selection of mappings $\tilde{h}(a)$ that can be applied in each step. We use the derivatives of the language K to find the appropriate letters a at each level of recursion. As previously, m limits the recursion depth.

$$\underline{E}_{L1}(Q, K, m) = \begin{cases} 0 & \text{if } Q \notin \mathcal{C} \text{ or } m = 0, \\ 1 + \max_{a \in K}\{\underline{E}_{L1}(Q \circ \tilde{h}(a), K//a, m-1)\} & \text{otherwise.} \end{cases}$$

$$\underline{E}_{L2}(Q, K, m) = \begin{cases} 0 & \text{if } Q \notin \mathcal{C} \text{ or } m = 0, \\ \max_{a \in K}\{res(Q, a)\} & \text{if } Q \in \mathcal{C},\ m > 0, \text{and } Q \circ \tilde{h}(a) \notin \mathcal{C} \text{ for all } a \in K, \\ 1 + \max_{a \in K}\{\underline{E}_{L2}(Q \circ \tilde{h}(a), K//a, m-1)\} & \text{otherwise.} \end{cases}$$

These formulae can be used for any language $K = \mathcal{P}(L^R)$, provided that L has the prefix property, as assumed in Theorem 2. The required operations on languages are particularly simple if L is regular. It can be then specified using a finite-state automaton, which reduces operations on infinite languages to the operations on their finite representations. Details are given in the following section.

7. The application of finite automata

We start by recalling the necessary notions of the theory of finite automata. For the original presentation see [19].

A *nondeterministic finite-state (Rabin-Scott) automaton* is a quintuplet:

$$\mathcal{M} = < V, S, s_0, T, I >,$$

where:

- V is an alphabet,
- S is a finite set of states,
- $s_0 \in S$ is a distinguished element of S, called the initial state,
- $T \subset S$ is a distinguished subset of S, called the set of final states,
- $I \subset V \times S \times S$ is a state transition relation.

We often write $(a, s_i) \rightarrow s_k$ instead of $(a, s_i, s_k) \in I$.

Finite state automata are commonly represented as directed graphs, with the nodes corresponding to states, and arcs representing transitions. The initial state is pointed to by a short arrow. The final states are distinguished by double circles.

A word $w = a_1 a_2 \ldots a_n \in V^*$ is *accepted* by the automaton $\mathcal{M}$ if there exists a sequence of states $s_0, s_1, s_2, \ldots, s_{n-1} \in S$ and $s_n \in T$ such that:

$$(a_1, s_0) \rightarrow s_1, \ (a_2, s_1) \rightarrow s_2, \ \cdots, \ (a_n, s_{n-1}) \rightarrow s_n.$$

Thus, w is accepted by $\mathcal{M}$ if there exists a directed path in the graph of $\mathcal{M}$ starting in the initial state s_0, ending

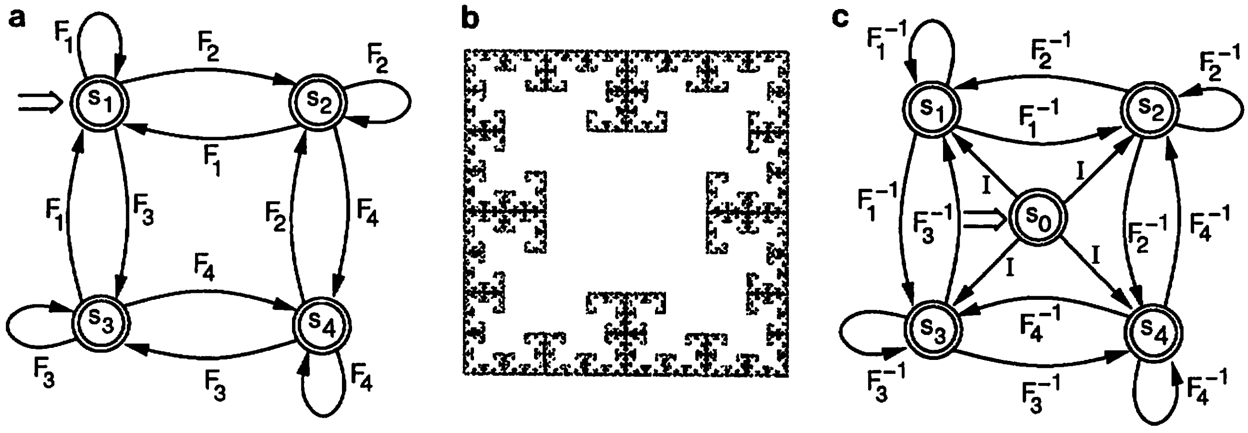


Figure 5: (a) The automaton $\mathcal{M}_1$ defining the language L_1, (b) the attractor of the LRIFS $\mathcal{I}_1$, and (c) the automaton $\mathcal{M}_1^{\mathcal{RP}}$ defining the language $K_1 = \mathcal{P}(L_1^R)$

in some final state s_n, and labeled with the consecutive letters of w. The set of all words accepted by an automaton $\mathcal{M}$ is called the *language accepted by* $\mathcal{M}$, and denoted by $L(\mathcal{M})$.

It is known that the mirror image of the language $L(\mathcal{M})$ is accepted by the automaton

$$\mathcal{M}^{\mathcal{R}} = \langle V, S \cup \{s_0'\}, s_0', \{s_0\}, I^R \rangle,$$

where $I^R =$

$$\{(\epsilon, s_0', s_k) : s_k \in T\} \cup \{(a, s_j, s_i) : (a, s_i, s_j) \in I\}.$$

Thus, the automaton $\mathcal{M}^{\mathcal{R}}$ is obtained from $\mathcal{M}$ by:

- creating a new initial state $s_0' \notin S$,
- creating transitions labeled ϵ from s_0' to all final states of $\mathcal{M}$,
- reversing the directions of all other transitions,
- making s_0 the unique final state of $\mathcal{M}^{\mathcal{R}}$.

Given an automaton $\mathcal{M}$ defining a language L, the prefix closure $\mathcal{P}(L^R)$ is accepted by the automaton $\mathcal{M}^{\mathcal{RP}}$ obtained from $\mathcal{M}^{\mathcal{R}}$ by making all its states final.

Consider an LRIFS $\mathcal{I} = \langle X, \mathcal{F}, V, h, L \rangle$, where L is accepted by a given finite automaton $\mathcal{M}$. Using the method given above, we can construct the automaton $\mathcal{M}^{\mathcal{RP}} = \langle V, S, s_0, S, I \rangle$ that accepts the language $K = \mathcal{P}(L^R)$. A word w belongs to K if and only if there exists a path in $\mathcal{M}^{\mathcal{RP}}$ starting in s_0 and labeled with the consecutive letters of w. Thus, the recursive computation of the derivatives of K, needed to evaluate functions E_{L1} and E_{L2}, can be replaced by the recursive construction of paths in $\mathcal{M}^{\mathcal{RP}}$, starting is s_0. This leads to the following recursive definitions:

$$\underline{E}_{M1}(Q, s_i, m) = \begin{cases} 0 & \text{if } Q \notin \mathcal{C} \text{ or } m = 0, \\ 1 + \max\limits_{(a,s_i,s_j) \in I} \{\underline{E}_{M1}(Q \circ \tilde{h}(a), s_j, m-1)\} & \\ & \text{otherwise.} \end{cases}$$

$$\underline{E}_{M2}(Q, s_i, m) = \begin{cases} 0 & \text{if } Q \notin \mathcal{C} \text{ or } m = 0, \\ \max\limits_{(a,s_i,s_j) \in I} \{res(Q, a)\} & \begin{array}{l}\text{if } Q \in \mathcal{C},\ m > 0, \text{and} \\ Q \circ \tilde{h}(a) \notin \mathcal{C} \\ \text{for all } (a, s_i, s_j) \in I,\end{array} \\ 1 + \max\limits_{(a,s_i,s_j) \in I} \{\underline{E}_{M2}(Q \circ \tilde{h}(a), s_j, m-1)\} & \\ & \text{otherwise.} \end{cases}$$

The evaluation of functions $\underline{E}_{M1}$ and $\underline{E}_{M2}$ starts with $s_i = s_0$. The equivalence of the formulae for $\underline{E}_{L1}$ and $\underline{E}_{M1}$, as well as $\underline{E}_{L2}$ and $\underline{E}_{M2}$, can be proved by induction on the maximum path length in $\mathcal{M}^{\mathcal{RP}}$.

Example 1. The following LRIFS $\mathcal{I}_1 = \langle X, \mathcal{F}_1, V_1, h_1, L_1 \rangle$ was described by Berstel and Abdallah [6]. It is assumed that:

- X is the plane,
- $\mathcal{F}_1$ consists of four transformations:

$$\begin{aligned} F_1 &= s(0.5) \circ t(0.0, 0.5), \\ F_2 &= s(0.5) \circ t(0.5, 0.5), \\ F_3 &= s(0.5), \\ F_4 &= s(0.5) \circ t(0.5, 0.0), \end{aligned}$$

- $V_1 = \{F_1, F_2, F_3, F_4\}$,
- $h_1(F_i) = F_i$ for $i = 1, 2, 3, 4$.

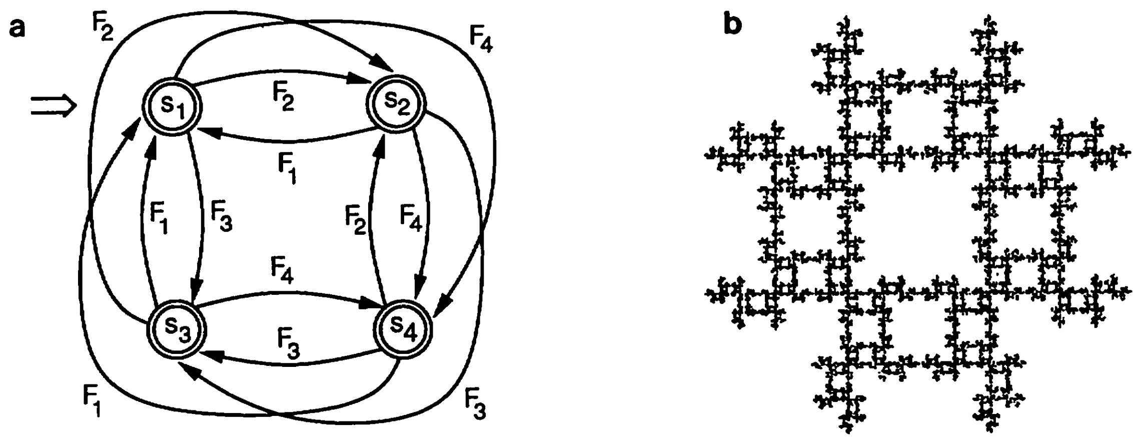


Figure 6: (a) The automaton $\mathcal{M}_2$ defining the language L_2, and (b) the attractor of the LRIFS $\mathcal{I}_2$

The language L_1 is defined using the finite automaton $\mathcal{M}_1$ shown in Figure 5a, and the corresponding attractor is given in Figure 5b. The automaton defining the language $K_1 = \mathcal{P}(L_1^R)$ is shown in Figure 5c, with the transitions labeled using the inverse transformations of $\mathcal{F}_1$. The label I indicates the identity transformation, associated with the ϵ-transitions of $\mathcal{M}_1^{\mathcal{RP}}$. Plate 1 (left) visualizes the escape time function computed with a recursion depth limit $m = 20$, using a bounding circle $\mathcal{C}$ with radius $R = 5$. The escape time values are interpreted as indices to a color map, arbitrarily divided into several ramps. Plate 1 (right) presents the same function as a height field.

Example 2. The LRIFS $\mathcal{I}_2$ considered in this example was described by Vrscay [20]. It uses the same set of transformations $\mathcal{F}$ and the labeling function h as $\mathcal{I}_1$, but the language L_2 is different. The automaton $\mathcal{M}_2$ defining L_2 and the resulting attractor are shown in Figure 6. The escape time function is presented in Plate 2.

Example 3. The LRIFS $\mathcal{I}_3$, taken from [16], describes a leaf-like structure with the alternating and opposite branches. The set of transformations is specified below:

$$\begin{aligned}
F_1 &= s(0.5) \circ t(-0.002, 0) \\
F_2 &= s(0.5) \circ t(0.002, 0) \\
F_3 &= s(0.5) \circ t(-0.002, 0.13) \\
F_4 &= s(0.5) \circ t(0.002, 0.13) \\
F_5 &= s(0.42) \circ r(45) \\
F_6 &= s(0.2) \circ r(90) \circ t(-0.05, 0.05) \\
F_7 &= s(0.2) \circ t(-0.05, 0.05) \\
F_8 &= t(0.3, -0.3) \circ s(0.74) \circ t(-0.3, 0.3) \\
F_9 &= s(0.37) \circ r(-45) \circ t(0, 0.14) \\
F_{10} &= s(0.172) \circ r(-90) \circ t(0.05, 0.19) \\
F_{11} &= s(0.172) \circ t(0.05, 0.19) \\
F_{12} &= t(-0.265, -0.405) \circ s(0.74) \circ t(0.265, 0.405) \\
F_{13} &= t(0, -1) \circ s(0.74) \circ t(0, 1)
\end{aligned}$$

The automaton $\mathcal{M}_3$ defining L_3 and the corresponding attractor are shown in Figure 7. The escape time function is visualized in Plate 3.

8. Conclusions

This paper presents methods for computing the escape-time functions of language-restricted iterated function systems. The LRIFS's generalize the ordinary IFS's by imposing restrictions on the applicable sequences of transformations. The escape-time functions can be computed for any set of sequences constituting a prefix-extensible formal language L. The computation of the escape time involves finding the mirror image L^R, determining the prefix language $K = P(L^R)$, and calculating its derivatives. These operations can be performed in a simple way if L is regular, using a specification of L by a finite automaton. All examples considered in this paper refer to this case. It is an open problem whether non-regular languages can yield other attractors and visualizations.

One could raise a question, whether this paper applies computer graphics to visualize an important mathematical concept, or whether it merely employs mathematics to create images for the sake of their visual appeal. Our motivation falls in both areas — we wanted to extend the mathematical concept of escape-time functions to LRIFS's, realizing that it is primarily used for image synthesis. In addition, we found that the well-established theory of automata and formal languages had unexpected applications in computer graphics.

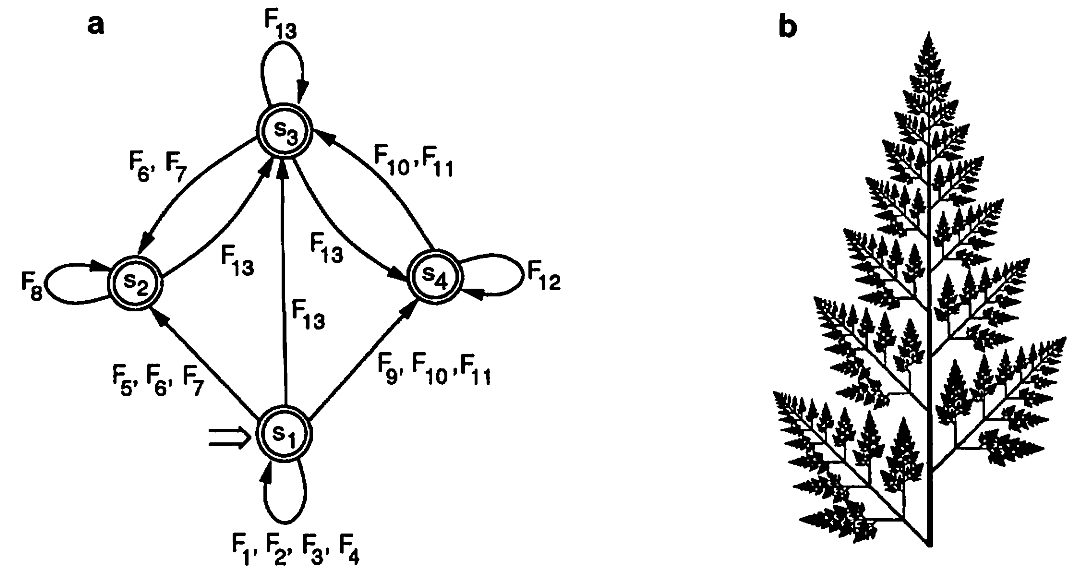


Figure 7: (a) The automaton $\mathcal{M}_3$ defining the language L_3, and (b) the attractor of the LRIFS $\mathcal{I}_3$

Appendix: Justification of the logarithmic formula for res(Z, a).

The continuity of the escape function can also be maintained by residual terms other than that given by Formula 1, for example:

$$\text{res}'(Z,a) = \begin{cases} \dfrac{R - \|Z\|}{\|Z \circ \tilde{h}(a)\| - \|Z\|} & \text{if } Z \in \mathcal{C} \text{ and } Z \circ \tilde{h}(a) \notin \mathcal{C}, \\ 0 & \text{otherwise.} \end{cases}$$

In order to explain the advantages of Formula 1, let us consider an IFS consisting of a single complex function $F(z) = z/c$. By definition, $F(z)$ is a contraction, thus $|c| > 1$. Given a circle $\mathcal{C}$ with radius R and center O in the origin of the coordinate system, the integer-valued escape-time function $E_1(z)$ is equal to:

$$E_1(z) = \max\{n \in \mathcal{N} : \|zc^n\| \leq R\}.$$

The symbol $\mathcal{N}$ represents the set of natural numbers (including zero), and the module of a complex number is identified with its norm, $|zc^n| = \|zc^n\|$. A continuous (and infinitely differentiable) extension of function $E_1(z)$ is:

$$E_2(z) = \max\{u \in \mathcal{R}^+ : \|zc^u\| \leq R\},$$

where $\mathcal{R}^+$ is the set of nonnegative real numbers. Obviously, the value $E_2(z)$ satisfies the equation:

$$\|zc^{E_2(z)}\| = R.$$

Consider point $Z = zc^{E_1(z)}$. By representing $E_2(z)$ as a sum $E_1(z) + \text{res}(Z)$, we obtain[3]:

$$\|zc^{E_1(z)+\text{res}(Z)}\| = \|Zc^{\text{res}(Z)}\| = R.$$

Note that $c = Zc/Z = F^{-1}(Z)/Z$, and take logarithms of both sides of the previous equation:

$$\log\|Z\| + \text{res}(Z)(\log\|F^{-1}(Z)\| - \log\|Z\|) = \log R.$$

Consequently,

$$\text{res}(Z) = \frac{\log R - \log\|Z\|}{\log\|F^{-1}(Z)\| - \log\|Z\|}.$$

Although the above reasoning applies to a particular IFS, it justifies the use of Function 1 also in other cases. In general, the distance between the origin of circle $\mathcal{C}$ and consecutive points in an escape trajectory tends to grow exponentially for large distance values. Consequently, Formula 1 minimizes first-order discontinuities in the escape-time function, yielding visually pleasing graphical representations.

Acknowledgements

We would like to thank Dr. Dietmar Saupe for the initial discussions of the concepts presented in this paper, the detailed comments of the manuscript, and for providing us with the program for rendering height fields.

[3]There is no need for specifying the second argument to the function res, as the IFS under consideration consists of a single transformation.

This research was sponsored by an operating grant and a graduate scholarship from the Natural Sciences and Engineering Research Council of Canada. The images were generated using facilities of the University of Calgary and the University of Regina.

References

[1] Ch. Bandt. Self-similar sets III. Constructions with sofic systems. *Monatsh. Math*, 108:89–102, 1989.

[2] M. F. Barnsley. *Fractals Everywhere.* Academic Press, 1988.

[3] M. F. Barnsley, J. H. Elton, and D. P. Hardin. Recurrent iterated function systems. *Constructive Approximation*, 5:3–31, 1989.

[4] M. F. Barnsley, A. Jacquin, F. Malassenet, L. Reuter, and A. D. Sloan. Harnessing chaos for image synthesis. Proceedings of SIGGRAPH '88 (Atlanta, Georgia, August 1-5, 1988). In *Computer Graphics*, 22, 4 (July 1988), pages 131–140, ACM SIGGRAPH, New York, 1988.

[5] M. A. Berger. Images generated by orbits of 2-D Markov chains. *Chance*, 2(2):18–28, 1989.

[6] J. Berstel and A. Nait Abdallah. Tétrarbres engendrées par des automates finis. Technical Report 89-7, Laboratoire Informatique Théorique et Programmation, Université P. et M. Curie, 1989.

[7] K. Culik II and S. Dube. Affine automata and related techniques for generation of complex images. Manuscript, University of South Carolina in Columbia, 1990.

[8] K. Culik II and S. Dube. Balancing order and chaos in image generation. Manuscript, University of South Carolina in Columbia, 1991.

[9] J.C. Hart. The object instancing paradigm for linear fractal modeling. In *Graphics Interface '92*, 1992. This Volume.

[10] J.C. Hart and T.A. DeFanti. Efficient antialiased rendering of 3-d linear fractals. Proceedings of SIGGRAPH '91 (Las Vegas, Nevada, July 28 – August 2, 1991). In *Computer Graphics*, 25, 4 (July 1991), pages 91–100, ACM SIGGRAPH, New York, 1991.

[11] D. Hepting, P. Prusinkiewicz, and D. Saupe. Rendering methods for iterated function systems. In H.-O. Peitgen, J.M. Heuriques, and L.F. Penedo, editors, *Fractals in the Fundamental and Applied Sciences*, pages 183–224, Amsterdam, 1991. North-Holland.

[12] J. E. Hutchinson. Fractals and self-similarity. *Indiana University Journal of Mathematics*, 30(5):713–747, 1981.

[13] B. B. Mandelbrot. *The Fractal Geometry of Nature.* W. H. Freeman, New York, 1982.

[14] H.-O. Peitgen and P. H. Richter, editors. *The Beauty of Fractals.* Springer-Verlag, Heidelberg, 1986.

[15] H.-O. Peitgen and D. Saupe, editors. *The Science of Fractal Images.* Springer-Verlag, New York, 1986.

[16] P. Prusinkiewicz and M. Hammel. Automata, languages, and iterated function systems. In J. C. Hart and F. K. Musgrave, editors, *Fractal Modeling in 3D Computer Graphics and Imagery*, pages 115–143. ACM SIGGRAPH, 1991. Course notes C14.

[17] P. Prusinkiewicz and A. Lindenmayer. *The Algorithmic Beauty of Plants.* Springer-Verlag, New York, 1990. With J. Hanan, F. D. Fracchia, D. R. Fowler, M. J. M. de Boer, and L. Mercer.

[18] P. Prusinkiewicz and G. Sandness. Koch curves as attractors and repellers. *IEEE Computer Graphics and Applications*, 8(6):26–40, November 1988.

[19] M. O. Rabin and D. Scott. Finite automata and their decision problems. *IBM J. Res. Develop.*, 3:114–125, 1959.

[20] E. R. Vrscay. Iterated function systems: Theory, applications and the inverse problem. In J. Bélair and S. Dubuc, editors, *Fractal Geometry and Analysis*, pages 405–468, Dordrecht, 1991. Kluwer Academic Pulishers.

[21] T. E. Womack. Linear and Markov iterated function systems in fractal geometry. Master's thesis, Virginia Polytechnic Institute and State University, Blacksburg, Virginia, 1989.

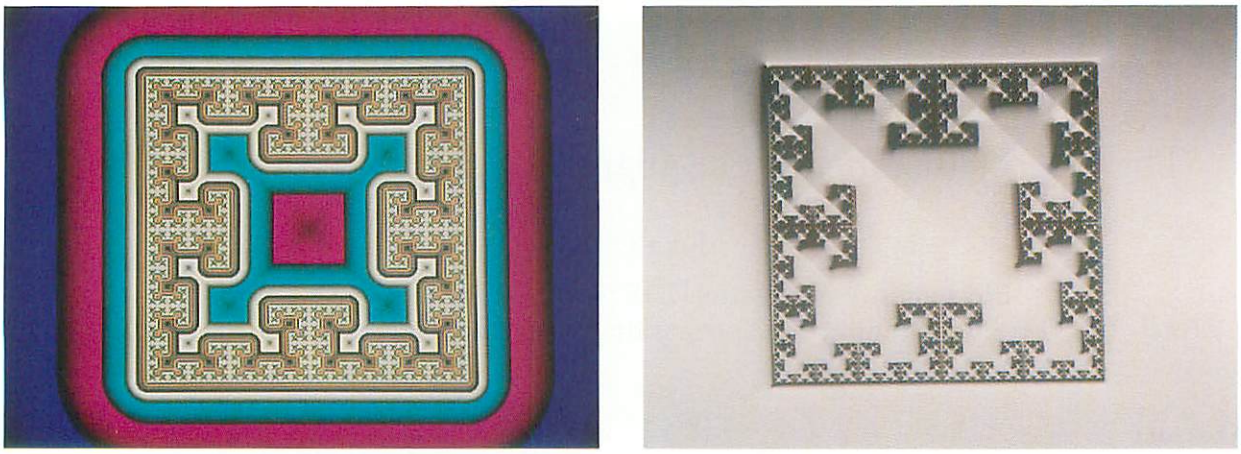


Plate 1: The escape time function for the attractor of the LRIFS $\mathcal{I}_1$, shown using a color map and as a height field

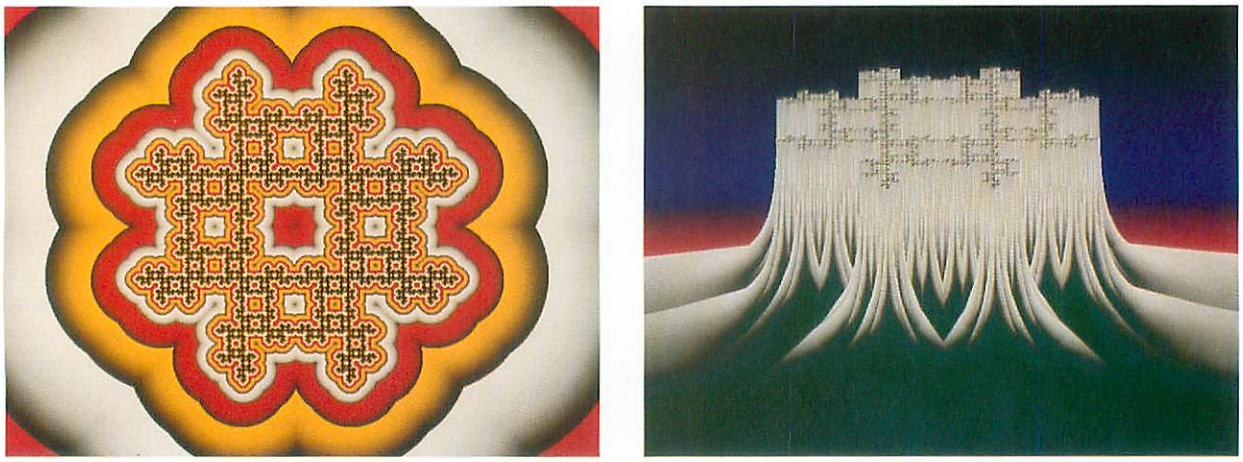


Plate 2: The escape time function for the attractor of the LRIFS $\mathcal{I}_2$, shown using a color map and as a height field

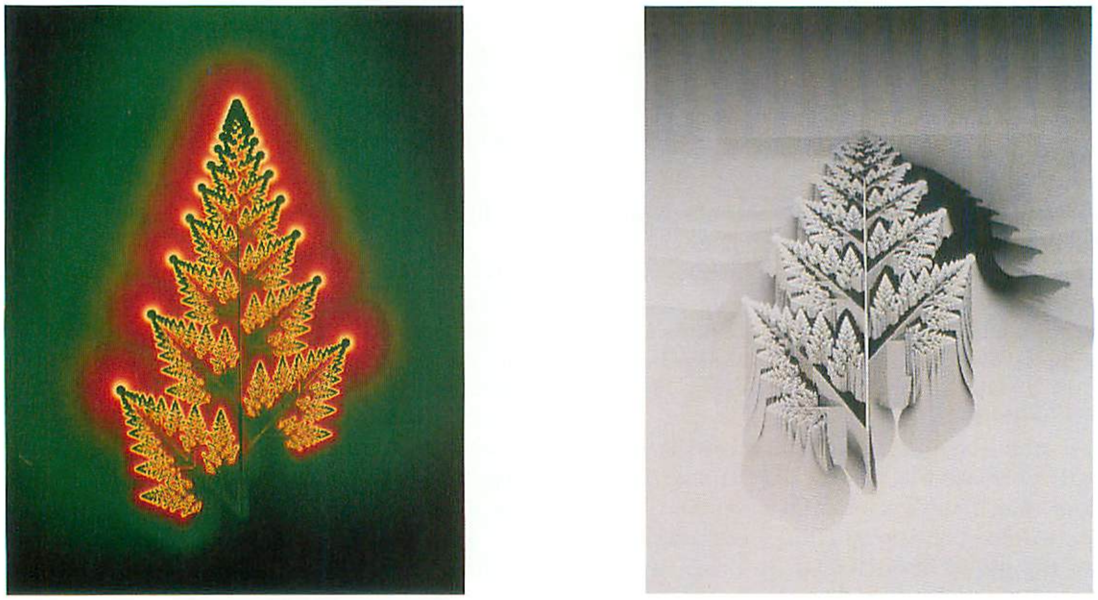


Plate 3: The escape time function for the attractor of the LRIFS $\mathcal{I}_3$, shown using a color map and as a height field

The Object Instancing Paradigm for Linear Fractal Modeling

John C. Hart

Electronic Visualization Lab, Univ. of Illinois at Chicago
Natl. Ctr. for Supercomputing Applications, Univ. of Illinois at Urbana-Champaign

Abstract

The recurrent iterated function system and the L-system are two powerful linear fractal models. The main drawback of recurrent iterated function systems is a difficulty in modeling whereas the main drawback of L-systems is inefficient geometry specification. Iterative and recursive structures extend the object instancing paradigm, allowing it to model linear fractals. Instancing models render faster and are more intuitive to the computer graphics community.

A preliminary section briefly introduces the object instancing paradigm and illustrates its ability to model linear fractals. Two main sections summarize recurrent iterated function systems and L-systems, and provide methods with examples for converting such models to the object instancing paradigm. Finally, a short epilogue describes a particular use of color in the instancing paradigm and the conclusion outlines directions for further research.

Keywords: Constructive Solid Geometry, L-system, Linear Fractal, Object Instancing, Recurrent Iterated Function System.

1 Introduction

Linear fractals are sets posessing some sort of self-affinity, with detail at all levels of magnification. Many of the examples used to demonstrate the basic concepts of fractal geometry are linear fractals, such as those decorating the first part of [Mandelbrot, 1982]. Other accounts have referred to linear fractals as "self-similar/self-affine sets" [Mandelbrot, 1982], "recurrent sets" [Dekking, 1982] and "graftals" [Smith, 1984].

The two most common linear fractal models are the recurrent iterated function system (RIFS) and the Lindenmeyer parallel graph grammar (L-system).

The iterated function system (IFS) model was used to investigate the use of linear fractals in image synthesis in [Demko *et al.*, 1985; Barnsley *et al.*, 1988]. Their rendering method used, often quite sophisticated, point clouds. An extension to the IFS model has been introduced and examined in various forms, under the adjectives "recurrent" [Dekking, 1982; Barnsley *et al.*, 1989; Cabrelli *et al.*, 1991], "Markov" [Womack, 1989], "controlled" [Prusinkiewicz & Lindenmayer, 1990], "language restricted" [Prusinkiewicz & Hammel, 1991] and "hierarchical" [Peitgen *et al.*, 1991]. Subtle differences in nomenclature and form have caused this heterogeneous development. Moreover, modeling objects using the RIFS model is difficult, and each of the above variations on the RIFS theme have distinct advantages for this task.

L-systems are parallel-graph grammars. Many have explored their applications in natural image synthesis, e.g. [Smith, 1984; Prusinkiewicz *et al.*, 1988; Prusinkiewicz & Lindenmayer, 1990], usually via the turtle graphics paradigm [Abelson & diSessa, 1982]. The L-system model provides a straightforward and powerful technique for modeling natural objects, though the geometry it produces using the turtle paradigm is not organized efficiently for rendering.

Both models, in many cases, can be simulated using the object instancing paradigm, which makes them more efficient for rendering. We begin with a review of the object instancing paradigm. Descriptions of the recurrent iterated function system and the L-system models follow, and each is completed with instructions for the translation of the model to the object instancing paradigm.

2 The Object Instancing Paradigm

Object instancing is a modeling technique that permits efficient internal representations of redundant objects. Often, complicated objects consist of many identical components. Object instancing capitalizes on the redundancy of these identical components. A good example of an ideal situation is seen in the animation "Megacycles" [Amanatides & Mitchell, 1989].

The object instancing paradigm consists of two basic kinds of objects: primitives and instances.

A *primitive* is an indivisible rendering atom. Boundary-representation systems use polygons or spline surfaces as primitives whereas solid modeling systems use natural quadrics, planes, tori and superquadrics as primitives.

An instance consists of a pointer to another object (a primitive or another instance, called the "master" object) and an affine transformation. The instance reproduces the referenced object, deformed by its affine transformation. [Sutherland, 1963].

An object may also be a CSG set-theoretic operation such as a union, intersection or difference. Their operands are pointers to other operations, and so, they too instance other objects. (For the purpose of this discussion, we will focus on the union operation.)

The object instancing paradigm contains another distinction: the global coordinate system where the final scene is assembled versus the local modeling coordinate systems where the canonical scene components are constructed. A canonical object is a shape in its default size, proportions and location. Canonical objects do not appear in the rendered scene. A final instancing operation is needed to transform the canonical objects from their local modeling-space coordinate systems into the global rendering-space coordinate system [Roth, 1982].

Primitives are canonical objects. The canonical sphere is the unit sphere centered at the origin; the canonical plane is the $y = 0$ plane intersecting the origin; the canonical cylinder is the cyclinder of unit radius and height centered and oriented along the interval $[0, 1]$ of the y-axis; the canonical cone is likewise defined, pointing in the positive y direction. One can render a canonical object in its default state by instancing it from canonical space into rendering space using the identity transformation matrix. When modeling, instancing takes canonical objects to canonical objects. Upon completion of the modeling process, an instance operation converts the final canonical object into a rendered object for image synthesis.

2.1 Examples

Figure 1 (left) shows an approximation to Sierpinski's gasket by nine triangles. Figure 1 (right) shows a possible CSG instancing model with a tree topology. The circles denote CSG union operations of three instances.

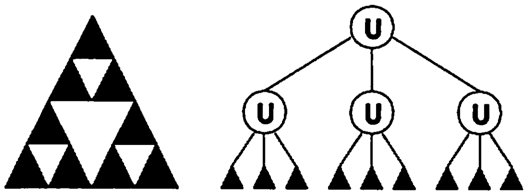


Figure 1: An approximation of Sierpinski's gasket (left) modeled with a tree topology (right).

In the absolute specification paradigm, each triangle is individually specified, commonly by its three endpoints. Further development of this tree-structured model of Sierpinski's gasket requires the specification of 3^{n-1} individual triangles, where n is the height of the tree. Hence, the size of an absolute specification model of a linear fractal grows exponentially as the detail increases.

The subtrees of Figure 1 (right) are redundant. Using the instancing paradigm, one creates the same gasket approximation, Figure 2 (left), using a graph topology, Figure 2 (right), with fewer nodes. Here, the small circles denote instancing operations.

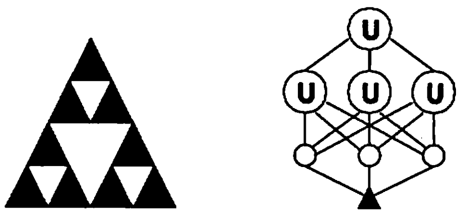


Figure 2: An approximation of Sierpinski's gasket (left) modeled with a graph topology (right).

The textual specification for such an instanced model might look like the following.

```
canonical object level_1_up
  triangle
  scale: 0.5, 0.5
  translate: 0.0, 0.5
}

canonical object level_1_left {
  triangle
  scale: 0.5, 0.5
  translate: -0.433, -0.25
}

canonical object level_1_right {
  triangle
  scale: 0.5, 0.5
  translate: 0.433, -0.25
}

canonical object level_2_up {
  union: level_1_up, level_1_left, level_1_right
  scale: 0.5, 0.5
  translate: 0.0, 0.5
}

canonical object level_2_left {
  union: level_1_up, level_1_left, level_1_right
  scale: 0.5, 0.5
  translate: -0.433, -0.25
}

canonical object level_2_right {
  union: level_1_up, level_1_left, level_1_right
  scale: 0.5, 0.5
  translate: 0.433, -0.25
}

object level_top {
  union: level_2_up, level_2_left, level_2_right
}
```

In the previous object instancing example there is no difference between a level two object such as level_2_up and a level one object such as level_1_up other than their name and the objects they instance. This is called iterative instancing and the size of its model grows linearly as the level of detail increases.

Instancing graphs are usually acyclic. A directed cycle in an instancing graph means an object has somehow instanced itself, and if rendered, may appear infinitely many times in the scene. This is called recursive instancing.

As shown in [Hart & DeFanti, 1991; Hart, 1991a], one can extend the object instancing paradigm to handle recursive instancing by culling the instancing process when a predefined global bounding volume become sufficiently small. This requires any instancing loop to be contractive, otherwise the instances would not converge.

For example, consider the limit of the shape described in Figures 1 and 2. Allowing recursive instancing, the hierarchy becomes cyclic and describes a linear fractal, specifically Sierpinski's gasket, in Figure 3.

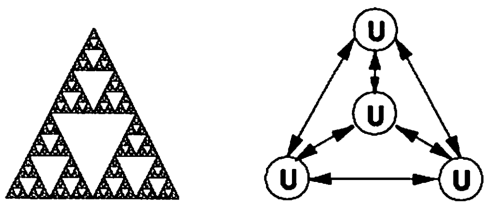


Figure 3: Sierpinski's gasket (left) modeled with a cyclic graph topology (right).

This linear fractal is specified, more tersely than before, with the following commands.

```
canonical object up {
        union: up, left, right
        scale: 0.5, 0.5
        translate: 0.0, 0.5
}

canonical object left {
        union: up, left, right
        scale: 0.5, 0.5
        translate: -0.433, -0.25
}

canonical object right {
        union: up, left, right
        scale: 0.5, 0.5
        translate: 0.433, -0.25
}

object top {
        union: up, left, right
}
```

A cyclic instancing graph represents infinitly high levels of detail. Hence, the size of a recursive instancing model remains constant as the level of detail increases.

3 The RIFS Model

A (hyperbolic) recurrent iterated function system $(\{w_i\}_{i=1}^N, G)$ consists of a finite set of N affine contractions w_i, and an N-vertex weakly-connected digraph G. Each vertex of this digraph corresponds to a contraction and each edge, an allowable contraction composition. Here, a digraph G is denoted by an ordered pair (G_v, G_e). The vertex set G_v is a set integers $\{1 \ldots N\}$. The edge set G_e is a set of ordered pairs $(i,j), i,j \in G_v$. such that the ordered pair $(i,j) \in G_e$ represents a directed edge from vertex i to vertex j.

3.1 The RIFS Attractor

Of fundamental importance to the study of iterated function systems as well as recurrent iterated function systems is the existence of corresponding unique compact non-empty invariant limit sets called attractors. One may recall the definition of the attractor of an IFS $\{w_i\}$ is the unique solution to the recurrence equation

$$A = \bigcup_{i=1\ldots N} w_i(A), \tag{1}$$

as originally shown in [Hutchinson, 1981]. The attractor of an RIFS $(\{w_i\}, G)$ is the union of the solution sets A_i to the recurrence equation

$$A_j = \bigcup_{(i,j)\in G_e} w_j(A_i) \tag{2}$$

as shown in [Barnsley *et al.*, 1989].

The attractor of an RIFS $(\{w_i\}_{i=1}^N, G)$ is a subset of the attractor of the IFS $\{w_i\}_{i=1}^N$. Thus, an RIFS is a restriction of an IFS, which results in the elimination of certain sections of attractor of the IFS. One example of this is the following set of four contractions

$$w_i(x,y) = \left(\frac{x \pm 1}{2}, \frac{y \pm 1}{2}\right) \tag{3}$$

taken over all combinations of signs. If G is a complete digraph of four vertices then the attractor of the RIFS $(\{w_i\}, G)$ is the attractor of the IFS $\{w_i\}$, namely the square $[-1,1] \times [-1,1]$. However, if G contains all edges except those of the type (i,i), then the same map may not be applied twice in a row. The attractor of this RIFS, from [Cabrelli *et al.*, 1991], is the fractal pound sign shown in Figure 4.

3.2 Modeling

One models an object as a linear fractal by creating a RIFS whose attractor approximates the object within some degree of accuracy. Equation (1) suggests that approximating an object with an IFS is as simple as finding a set of contractions that take the entire object to each

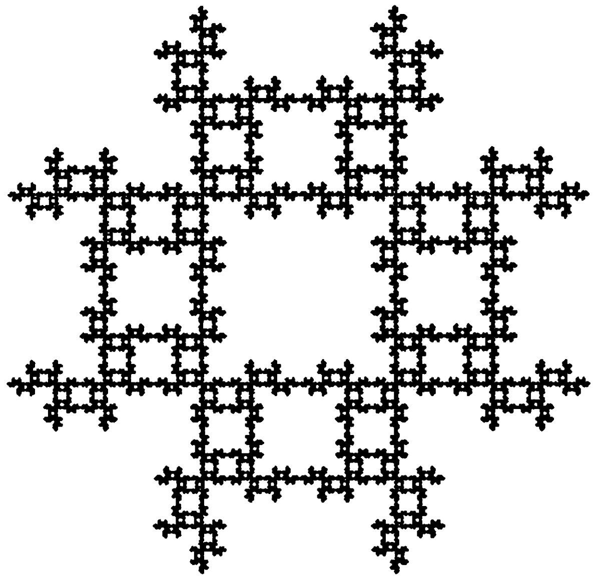


Figure 4: The fractal pound sign.

of its components — the so-called "collage theorem" philosophy of modeling [Barnsley *et al.*, 1986].

Equation (2) suggests a similar property, though for an RIFS model, the object is simulated by finding contractions that take parts of the object to smaller parts. In the previous example, the fractal pound sign consists of four images of three-fourths of the original pound sign. One can see the self-similarity of the fractal pound sign by comparing its first quadrant with the rest of the set.

Ordinarily, this partial self-similarity is quite difficult to see in objects. The collage theorem philosophy of modeling requires a self-tiling, which can be easily visualized. Its recurrent form, from [Barnsley *et al.*, 1989], requires a partial self-tiling which is significantly more difficult and, at best, non-intuitive. Hence, the recurrent collage theorem is a somewhat ineffective modeling tool for linear fractals, for which there are, unfortunately, few alternatives.

3.3 Specification via Instancing

The recursive object instancing paradigm is one alternative modeling method to the collage theorem. The drawback is one must still partially self-tile an object to create a linear fractal model of it. The benefit is the paradigm incorporates tools familiar to the computer graphics community.

Converting an RIFS to an instancing structure requires three simple steps. First N canonical instancing objects are constructed. Each affine contraction of the RIFS becomes the affine transformation matrix of its corresponding instancing object. The control digraph of the RIFS dictates the masters of each instance. Finally, a rendered object instances the canonical instances along with an initial bounding volume into rendering space.

For example, the 3-D pound sign can be specified by the following commands. The canonical object's names are abbreviated versions of the instance's position, from lower-left-rear to upper-right-front.

```
canonical object llr {
  union: llf, lrr, lrf, ulr, ulf, urr, urf
  scale: 0.5, 0.5, 0.5
  translate: -0.5, -0.5, -0.5
}

canonical object llf {
  union: llr, lrr, lrf, ulr, ulf, urr, urf
  scale: 0.5, 0.5, 0.5
  translate: -0.5, -0.5, 0.5
}
         .
         .
         .
canonical object urf {
  union: llr, llf, lrr, lrf, ulr, ulf, urr
  scale: 0.5, 0.5, 0.5
  translate: 0.5, 0.5, 0.5
}

object pound_sign {
  union: llr, llf, lrr, lrf, ulr, ulf, urr, urf
}
```

The object this instancing text models is shown in Figure 5.

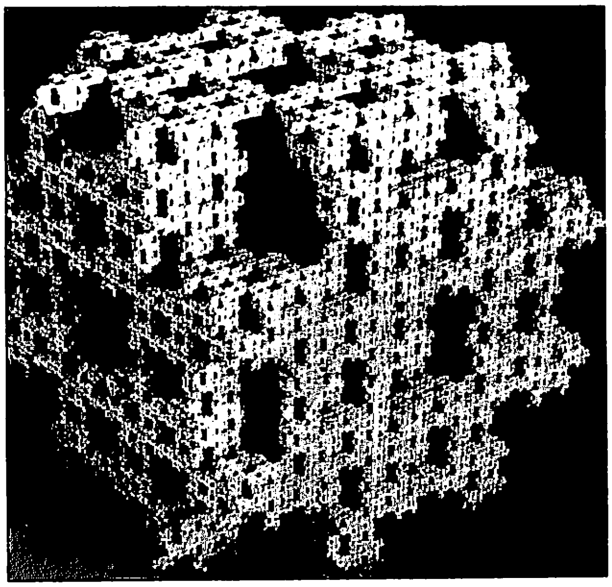


Figure 5: The 3-D fractal pound sign.

4 The L-System Model

Whereas the RIFS resembles the finite state automaton, an L-system is a parallel graph grammar similar in design to a context-free grammar (CFG). It consists of an alphabet, a set of symbols, an initial axiom and a set of productions. CFG productions are applied one at a time and one per step, usually to the leftmost symbol whereas L-system productions are applied in parallel — all symbols are replaced simultaneosly at each step.

4.1 Turtle Geometry

The words of an L-system are commonly interpreted, using turtle geometry [Abelson & diSessa, 1982], as a graphical object. The alphabet for turtle control contains such symbols as F which draws a straight line, and $+$ and $-$ which turn the turtle left and right by some fixed angle. For example, if this fixed angle is 90°, then the word

$$F+F-F-F+F-F-F+F-F-F+F-F \quad (4)$$

will draw the outline of a plus sign. In [Prusinkiewicz & Lindenmayer, 1990], a complete parametric language is used to developed a sophisticated 3-D turtle graphics paradigm.

Of particular utility are the symbols [and]. The left bracket pushes the current state of the turtle on a stack whereas the right bracket pops this stack, restoring the turtle to its previous state. This simulates branching structures, allowing the turtle to concentrate on an intricate branch without having to retrace its steps to get back to the base of the branch to draw the rest of the object.

For example, the following parametric L-system generates a ternary-branching tree, from Figure 2.8(a) in [Prusinkiewicz & Hammel, 1991],

$$\begin{aligned} A &\rightarrow\ !(\tfrac{\sqrt{3}}{3})\ F(50)\ [\&(18.95)\ F(50)\ A] \\ &\quad\ /(94.74)\ [\&(18.95)\ F(50)\ A] \\ &\quad\ /(132.63)\ [\&(18.95)\ F(50)\ A] && (5) \\ F(l) &\rightarrow F(1.109l) && (6) \\ !(w) &\rightarrow\ !(\sqrt{3}w) && (7) \end{aligned}$$

where $!(\cdot)$ alters the width of lines (radius of cylinders) by the given factor and $F(\cdot)$ draws a line of the given length. The rotation symbol $\&(\cdot)$ alters the pitch of the turtle; the symbol $/(\cdot)$ alters the roll. This L-system begins with the word $!(1)\ F(50)\ A$.

In the previous example, the shortest length of any branch is 50. We can convert this L-system from "enlongating nodes" to "deacreasing apices," which makes the longest length of any branch 50. This produces the following L-system, a recurrent string, which is equivalent to the previous one except for scale,

$$\begin{aligned} A(w,l) &\rightarrow\ !(w)\ F(l)\ [\&(18.95)\ F(l)\ A(\tfrac{w}{\sqrt{3}}, \tfrac{l}{1.109})] \\ &\quad\ /(94.74)\ [\&(18.95)\ F(l)\ A(\tfrac{w}{\sqrt{3}}, \tfrac{l}{1.109})] \\ &\quad\ /(132.63)\ [\&(18.95)\ F(l)\ A(\tfrac{w}{\sqrt{3}}, \tfrac{l}{1.109})]. \end{aligned} \quad (8)$$

This L-system begins with the initial word $!(1)\ F(50)\ A(0.577, 45.1)$.

4.2 Specification via Instancing

Conversion from an L-system to an object instancing structure is more difficult than converting to an RIFS. In [Prusinkiewicz & Lindenmayer, 1990], steps were described for converting an L-system model to an RIFS model. Hence, using this result with the last section, we could convert an L-system indirectly to an instancing structure. Instead, we will outline techniques for directly converting L-system models to object instancing models.

In general, converting L-system productions to instances requires thorough knowledge of the current state of the turtle at each step in the production. With the exception of the left bracket, each of the symbols on the right-hand side of the productions affect the state of the turtle. Furthermore, the line-drawing symbols cause a geometric addition to the scene as well as a change to the state of the turtle.

Each line-drawing symbol, e.g. F, can be interpreted as an instance of some canonical 3-D line, such as a cylinder. The cumulative effect of the symbols preceding the line-drawing symbol determine that line's size, position and orientation. Any bracket delimited symbols can be ignored. The affine transformation matrix of the line-instance is the product of the transformation matrices associated with the symbols preceding the current line-drawing symbol. For example, a $+$ corresponds to a rotation and an F (preceding the current symbols) corresponds to a translation.

Symbols denoting other productions are particularly difficult. These symbols can be replaced by an equivalent sequence of symbols denoting the production's cumulative affect on the turtle's state. Assessment of this cumulative affect can be quite cumbersome, particularly when the production is recursive.

Fortunately, in most botanical models, recursion is used to simulate branching patterns. Rather than having the turtle retracing its steps back to the branch root after drawing the branch, the branch production is bracketed. Hence, if a bracketed branch production symbol precedes the current line-drawing symbol, it can be ignored and its cumulatice affect on the turtle's state need not be computed.

The following instancing structure approximates the L-system (8).

```
canonical object trunk {
  cylinder
}

canonical object A {
  union: trunk, branch_1, branch_2, branch_3
  scale: 0.902, 0.902, 0.902
  translate: 0, 1, 0
}

canonical object branch_1 {
  union: trunk, A
  rotate: x,18.95
  translate: 0, 1, 0
}

canonical object branch_2 {
  union: trunk, A
  rotate: x, 18.95
  rotate: y, 94.74
  translate: 0, 1, 0
}

canonical object branch_3 {
  union: trunk, A
  rotate: x, 18.95
  rotate: y, 94.74
  rotate: y, 132.63
  translate: 0, 1, 0
}

object tree {
  union: trunk, A
}
```

This approximation of the L-system suffers from two major shortcomings.

First, the branch width changing factor is not represented in this model; the scaling transformation in the object named *A* is uniform. One might be tempted to use the scaling command: "scale: 0.577, 0.902, 0.577," which would produce a properly proportioned branch segment, but the rest of its sub-branches would be artificially skewed along the original branch's major axis.

Second, this L-system will be evaluated to the pixel level, producing very fine branches but no leaves. A more realistic model would terminate well before the limit structure, allowing the addition of leaves to the ends of the branches.

Both of these problems are solved by duplicating the above instancing text several times, into levels. This is equivalent to the use of conditions in a parametric L-system. The resulting instancing structure might look like the following, in a parametric instancing scheme.

```
canonical object A(0) {
  union: leaf
  translate: 0, 1, 0
}

canonical object A(n) {
  union: trunk(11-n), branch_1(n),
         branch_2(n), branch_3(n)
  scale: 0.902, 0.902, 0.902
  translate: 0, 1, 0
}

canonical object branch_1(n) {
  union: trunk(11-n), A(n-1)
  rotate: x,18.95
  translate: 0, 1, 0
}

canonical object branch_2(n) {
  union: trunk(11-n), A(n-1)
  rotate: x, 18.95
  rotate: y, 94.74
  translate: 0, 1, 0
}

canonical object branch_3(n) {
  union: trunk(11-n), A(n-1)
  rotate: x, 18.95
  rotate: y, 94.74
  rotate: y, 132.63
  translate: 0, 1, 0
}

canonical object trunk(0) {
  instance: cylinder
}

canonical object trunk(n) {
  instance: trunk(n-1)
  scale: 0.64, 1, 0.64
}

object tree {
  union: trunk(0), A(10)
}
```

In this case, the factor 0.64 unscales the radius of each branch by the value 0.902, then rescales it by the proper value 0.577 without affecting the geometry of its sub-branches.

The above parametric instancing specification can be implemented for standard instancing interpreters by enumerating each object for all possible values of *n*, incorporating the value *n* into the name of the object. For example, instead of `trunk(8)` instancing `trunk(7)`, the object `trunk_level_8` would instance `trunk_level_7`. The latter is significantly more verbose, but is still much less than absolute specification of every branch. In fact, parametric instancing specifies in constant steps what standard instancing specifies in logarithimic steps — what absolute specification specifies in linear steps.

A similar L-system was used to model the tree in Figure 6 (compare the trees in [Kay & Kajiya, 1986]). The branches are iteratively instanced cylinders, but the leaves are modeled by a recursive instancing structure derived from the IFS leaf model shown in [Demko *et al.*, 1985]. The grass is modeled as iteratively instanced as in [Snyder & Barr, 1987], though the grass blades here are cones.

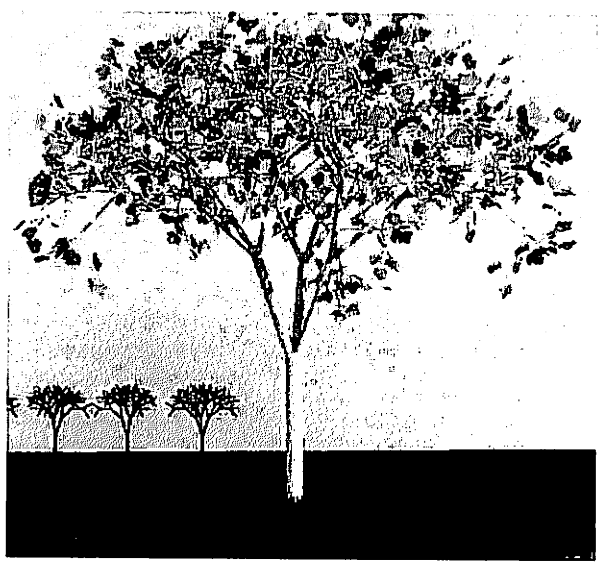


Figure 6: Autumn — an L-system tree with IFS leaves on instanced grass.

5 Incorporating Color

Color is incorporated into the linear fractal instancing model by associating a color and a weight with each instance. The color is specified by an RGB triple, the weight, by an alpha value. A current color and alpha value are maintained for the object during CSG hierarchy traversal. During rendering, when an operand is instanced, the instance tints the color of its parent in the hierarchy by its own individual color.

Formally, this tinting is based on the transfer functions from image compositing [Porter & Duff, 1984],

$$\begin{aligned} \mathbf{c} &= \mathbf{c}_o + \alpha_i(1-\alpha_o)\mathbf{c}_i, & (9) \\ \alpha &= \alpha_o + \alpha_i(1-\alpha_o), & (10) \end{aligned}$$

where $\mathbf{c}, \alpha$ are the resulting color and alpha values, $\mathbf{c}_p, \alpha_p$ are those of the parent, and $\mathbf{c}_i, \alpha_i$ are the tinting values of the instance.

The eight instances used to generate Figure 5 had the eight associated colors: black, red, green, yellow, blue, magenta, cyan and white, each with an alpha value of one-half. This coloring scheme was also used in [Hart & Das, 1991; Hart, 1991b] to better clarify otherwise imperceptible features of highly intricate linear fractal surfaces.

The coloring of the elm trees in the Fractal Forest [Hart & DeFanti, 1991; Hart, 1991a] was carefully adjusted so that the trunk and most of the visible branches were brown but the leaves (actually tiny branches) were green. Assigning the trunk a brown tint with a high alpha value and the branches a green color with a low alpha value achieved this goal. Thus, once a trunk instance was used, all of its decendents would be mostly brown whereas green points in the set arose from almost exclusive application of branch instances.

6 Conclusion

With a few enhancements, the standard instancing paradigm can be adapted to model linear fractals. This paper discussed methods for converting two popular linear fractal models to the instancing paradigm, for efficient implementation in current computer graphics systems. Several examples were given, with modeling code and resulting images. Additionally, the use of color in linear fractal models was documented.

6.1 Further Research

As they become better understood, linear fractals are becoming more popular for their applications in natural modeling and image synthesis. A few problems still remain regarding their rendering, such as a rigorous treatment of the shading properties of a fractal surface [Hart, 1991a].

Currently, the most popular open problem regarding the modeling of linear fractals is the algorithmic solution to the inverse problem: given a shape, automatically determine the parameters of a linear fractal model that simulates the shape within a prescribed accuracy. One solution uses a gradient search in the RIFS parameter space [Vrscay & Roehrig, 1989], but this method is prone to local minima traps. Further enhancement using genetic programming techniques appear promising [Vrscay, 1991], but much work remains along this specific course of attack.

One of the immediate applications of an automated solution to the inverse problem is image compression. Toward this end, block coding methods have been developed to solve this particular application of the inverse problem [Jacquin, 1991]. Nonetheless, image compression is only one tiny application of what could be a very powerful result. The ability to automatically model intricate 3-D structures with linear fractals would be a major result in computer graphics — its research should not end with one simple solution geared toward an image compression application.

Finally, the classic object instancing paradigm is not fully suited to modeling many of the subtle enhancements to the turtle-geometric L-system model, such as tropism [Prusinkiewicz & Lindenmayer, 1990] or stochastic variations [Kajiya, 1983; Bouville, 1985]. More research on extensions to the object instancing paradigm would allow more efficient modeling and rendering of more realistic botanical structures.

6.2 Acknowledgements

Tom DeFanti and Larry Smarr arranged the funding of this research, through a postdoctoral grant from the National Science Foundation.

The author performed this research on an AT&T Pixel Machine 964dX and a 940dX and is grateful to AT&T for their equipment donations.

Finally, the author would like to thank John Amanatides, Don Mitchell, Daryl Hepting, Arnaud Jacquin, Przemyslaw Prusinkiewicz and Ed Vrscay for their communication.

References

Abelson, H. and diSessa, A. A. *Turtle Geometry.* MIT Press, 1982.

Amanatides, J. and Mitchell, D. P. Megacycles. *SIGGRAPH Video Review*, 51, 1989. (Animation).

Barnsley, M. F., Ervin, V., Hardin, D., and Lancaster, J. Solution of an inverse problem for fractals and other sets. *Proceedings of the National Academy of Science*, 83:1975–1977, April 1986.

Barnsley, M. F., Jacquin, A., Mallassenet, F., Rueter, L., and Sloan, A. D. Harnessing chaos for image synthesis. *Computer Graphics*, 22(4):131–140, 1988.

Barnsley, M. F., Elton, J. H., and Hardin, D. P. Recurrent iterated function systems. *Constructive Approximation*, 5:3–31, 1989.

Bouville, C. Bounding ellipsoids for ray-fractal intersection. *Computer Graphics*, 19(3):45–51, 1985.

Cabrelli, C., Molter, U., and Vrscay, E. R. Recurrent iterated function systems: Invariant measures, a collage theorem and moment relations. In *Proceedings of the First IFIP Conference on Fractals.* Elsevier, 1991.

Dekking, F. M. Recurrent sets. *Advances in Mathematics*, 44:78–104, 1982.

Demko, S., Hodges, L., and Naylor, B. Construction of fractal objects with iterated function systems. *Computer Graphics*, 19(3):271–278, 1985.

Hart, J. C. and Das, S. Sierpinski blows his gasket. *SIGGRAPH Video Review*, 61, 1991. (Animation).

Hart, J. C. and DeFanti, T. A. Efficient antialiased rendering of 3-D linear fractals. *Computer Graphics*, 25(3), 1991.

Hart, J. C. *Computer Display of Linear Fractal Surfaces.* PhD thesis, EECS Dept., University of Illinois at Chicago, Sept. 1991.

Hart, J. C. unNatural Phenomena. *SIGGRAPH Video Review*, 71, 1991. (Animation).

Hutchinson, J. Fractals and self-similarity. *Indiana University Mathematics Journal*, 30(5):713–747, 1981.

Jacquin, A. E. Image coding based on a fractal theory of iterated contractive image transformations. In Hart, J. C. and Musgrave, F. K., editors, *Fractal Models in 3-D Computer Graphics and Imaging*, pages 245–270. ACM SIGGRAPH '91 (Course #14 Notes), 1991.

Kajiya, J. T. New techniques for ray tracing procedurally defined objects. *ACM Transactions on Graphics*, 2(3):161–181, 1983. Also appeared in *Computer Graphics 17*, 3 (1983), 91-102.

Kay, T. L. and Kajiya, J. T. Ray tracing complex scenes. *Computer Graphics*, 20(4):269–278, 1986.

Mandelbrot, B. B. *The Fractal Geometry of Nature.* W.H. Freeman, San Francisco, 2nd edition, 1982.

Peitgen, H.-O., Jurgens, H., and Saupe, D. *Fractals for the Classroom.* Springer-Verlag, New York, 1991.

Porter, T. and Duff, T. Compositing digital images. *Computer Graphics*, 18(3):253–259, 1984.

Prusinkiewicz, P. and Hammel, M. Automata, languages and iterated function systems. In Hart, J. C. and Musgrave, F. K., editors, *Fractal Models in 3-D Computer Graphics and Imaging*, pages 115–143. ACM SIGGRAPH '91 (Course #14 Notes), 1991.

Prusinkiewicz, P. and Lindenmayer, A. *The Algorithmic Beauty of Plants.* Springer-Verlag, New York, 1990.

Prusinkiewicz, P., Lindenmayer, A., and Hanan, J. Developmental models of herbaceous plants for computer imagery purposes. *Computer Graphics*, 22(4):141–150, August 1988.

Roth, S. D. Ray casting for modeling solids. *Computer Graphics and Image Processing*, 18(2):109–144, February 1982.

Smith, A. R. Plants, fractals, and formal languages. *Computer Graphics*, 18(3):1–10, July 1984.

Snyder, J. M. and Barr, A. H. Ray tracing complex models containing surface tessellations. *Computer Graphics*, 21(4):119–128, 1987.

Sutherland, I. E. Sketchpad: A man-machine graphical communication system. *Proceedings of the Spring Joint Computer Conference*, 1963.

Vrscay, E. R. and Roehrig, C. J. Iterated function systems and the inverse problem of fractal construction using moments. In Kaltofen, E. and Watt, S. M., editors, *Computers and Mathematics*, pages 250–259, New York, 1989. Springer-Verlag.

Vrscay, E. R. Iterated function systems: Theory, applications and the inverse problem. In *Lectures of the NATO Advanced Study Institute on Fractal Geometry and Analysis*, Montreal, 1991. Kluwer.

Womack, T. E. Linear and markov iterated function systems in fractal geometry. Master's thesis, Virginia Polytechnic Institute, May 1989.

Algorithms for Intersecting Parametric and Algebraic Curves

Dinesh Manocha
Computer Science Division
University of California at Berkeley
Berkeley, CA 94720, USA

James Demmel
Computer Science Division
and Mathematics Department
University of California at Berkeley
Berkeley, CA 94720, USA

Abstract

The problem of computing the intersection of parametric and algebraic curves arises in many applications of computer graphics, geometric and solid modeling. Earlier algorithms are based on techniques from Elimination theory or subdivision and iteration. The former is however, restricted to low degree curves. This is mainly due to issues of efficiency and numerical stability. In this paper we use Elimination theory and express the resultant of the equations of intersection as a matrix determinant. The matrix itself rather than its symbolic determinant, a polynomial, is used as the representation. The algorithm for intersection corresponds to substituting the other equation to construct an equivalent matrix such that the intersection points can be extracted from the eigenvalues and eigenvectors of the latter. Moreover, the algebraic and geometric multiplicities of the eigenvalues give us information about the intersection (multiplicity, tangential intersection etc.). As a result we are able to accurately compute higher order intersections. The main advantage of this approach lies in its *efficiency and robustness*. Moreover, the numerical accuracy of these operations is well understood. For almost all cases we are able to compute accurate answers in 64 bit IEEE floating point arithmetic.

Keywords: Intersection, Curves, Eigenvalues, Accuracy, Robustness.

1. Introduction

The problems of computing the intersection of parametric and algebraic curves are fundamental to geometric and solid modeling. Parametric curves, like B-splines and Bézier curves, are extensively used in the modeling systems and algebraic plane curves are becoming popular as well [Hof89, MM89, SP86, Sed89]. Intersection is a primitive operation in the computation of a boundary representation from a CSG (constructive solid geometry) model in a CAD system. Other applications of intersection include hidden curve removal for free form surfaces, [EC90]. Algorithms for computing the intersection of these curves have been extensively studied in the literature.

As far as computing the intersection of rational parametric curves is concerned, algorithms based on implicitization [Sed83], Bézier subdivision [LR80] and interval arithmetic [KM83] are well known. The implicitization approach computes the implicit form of one of the curve using resultants [Sed83]. Given the implicit representation of one curve, substitute the second parametrization and obtain a univariate polynomial in its parameter. The problem of intersection corresponds to computing the roots of the resulting polynomial. The Bézier subdivision relies on the convex hull property of Bézier curves and de Casteljau algorithm for subdividing Bézier curves. The resulting algorithm performs a linearly converging binary search. It has been improved by more effective use of the convex hull property [SWZ89]. The interval arithmetic uses an idea similar to subdivision. Each curve is preprocessed to determine its vertical and horizontal tangents, and the curve is divided into 'intervals' which have vertical or horizontal tangents only at the endpoints.

The relative performance and accuracy of these algorithms is highlighted in [SP86]. In particular, implicitization based approaches are faster as compared to other methods for curves of degree up to four. However, their relative performance degrades for higher degree curves. This is mainly due to issues of numerical stability and their effect on the choice of representation and algorithms for root finding. For curves of degree greater than three, the resulting univariate polynomial has degree 16 or higher. The problem of computing real roots of such high degree polynomials can be ill-conditioned [Wil59]. Using exact arithmetic or symbolic techniques to overcome the numerical problems have a considerable effect on the efficiency of the problem and therefore, subdivision based algorithms perform better.

The algorithms for algebraic curve intersection are analogous to those of intersecting parametric curves. Resultants can be used to eliminate one variable from the two equations corresponding to the curves. The problem of intersection corresponds to computing roots of the resulting univariate polynomial. This approach causes numerical problems for higher degree curves (greater than four). A robust algorithm based on subdivision has been presented in [Sed89]. However, resultant based algorithms are considered to be the fastest for lower degree curves.

In many applications, the intersection may be of higher order involving tangencies and singular points. Such instance are rather common in industrial applications [MM89]. Most algorithms require special handling for tangencies and thereby requiring additional computation for detecting them. In fact algorithms based on subdivision and Newton–type techniques often fail to accurately compute the intersections in such cases. Special techniques for computing first order tangential contacts of parametric curves are given in [MM89]. [Sed89] presents modification of his algorithm for computing all double points of an algebraic curve in a triangular domain. However, no efficient and accurate techniques are known for computing higher order intersections for general cases.

In this paper we present efficient and robust algorithms for intersecting parametric and algebraic curves. For parametric curves we implicitize one of the curves and represent the implicit form as a *matrix determinant.* However, we do not compute the symbolic determinant and express the implicit formulation as a matrix. The idea of matrix determinant has been used in [MC91] to represent and evaluate the intersection of rational parametric surfaces. Given the implicit form, we substitute the other parametrization into the matrix formulation and use the resulting matrix to construct a numerical matrix such that the intersection points can be computed from its eigendecompositon. This is in contrast with expanding the symbolic determinant and finding the roots of the resulting polynomial. The advantages of this technique lie in ***efficiency, robustness and numerical accuracy***. The algorithms for computing eigenvalues and eigenvectors of a matrix are *backward stable* and fast implementations are available as part of packages like EISPACK and LAPACK [GL89, ABB$^+$90]. Furthermore, we effectively use the algebraic and geometric multiplicities of the eigenvalues to determine the exact multiplicity of the intersection. The algorithm for intersecting algebraic curves is rather similar, except the relationship between algebraic and geometric multiplicities of the eigenvalue and the multiplicity of intersection is different.

The rest of the paper is organized in the following manner. In section 2 we present our notation and review techniques from Elimination theory for implicitizing parametric curves. Furthermore, we show that the problems of intersecting parametric and algebraic curves can be reduced to computing roots of polynomials expressed as matrix determinants. In section 3, we present results from linear algebra and numerical analysis being used in the algorithm. Section 4 deals with reducing the problem of root finding to computing the eigendecomposition. Given the eigenvalues and eigenvectors, we compute the intersection points of parametric curves in the domain of interest. We also discuss the performance and robustness of the resulting algorithm. Section 5 deals with higher order intersections and illustrates the technique with examples.

2. Parametric and Algebraic Curves

A rational Bézier curve is of the form [BBB87]:

$$\mathbf{P}(t) = (X(t), Y(t)) = \frac{\Sigma_{i=0}^{n} w_i \mathbf{P}_i B_{i,n}(t)}{\Sigma_{i=0}^{n} w_i B_{i,n}(t)}, \qquad 0 \leq t \leq 1$$

where $\mathbf{P}_i = (X_i, Y_i)$ are the coordinates of a control point, w_i is the weight of the control point and $B_{i,n}(t)$ corresponds to the Bernstein polynomial, $B_{i,n} = \binom{n}{i}(1-t)^{n-i}t^i$. Rational curves like B-splines can be converted into a series of Bézier curves by knot insertion algorithms [BBB87]. Thus, the problem of intersecting rational curves can be reduced to intersecting Bézier curves. Each of these curves is described by its corresponding control polygon and the curve is always contained in the convex hull of the control points. Therefore, the intersection of the convex hull of two such curves is a necessary condition for the intersection of curves. One such instance has been highlighted in Fig. I.

Algebraic plane curves are generally expressed in standard power basis:

$$F(x,y) = \Sigma_{i+j \leq n} c_{ij} x^i y^j = 0.$$

They can also be represented in Bernstein basis. The problem of intersection corresponds to computing the common points on such curves in a particular domain.

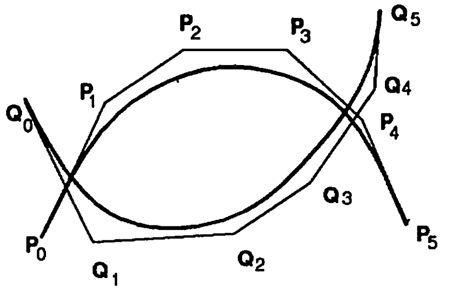


Fig. I
Intersection of Bézier curves

2.1. Resultants

Given two polynomials in one unknown, their resultant is a polynomial in their coefficients. Moreover, the vanishing of the resultant is a necessary and sufficient condition for the two polynomials to have a common root. Three methods are known in the literature for computing the resultant, owing to Sylvester, Bezout and Cayley [Sal85].

Each of them expresses the resultant as determinant of a matrix. The order of the matrix is different for different methods. We use Cayley's formulation as it results in a matrix of minimum order.

Given two polynomials, $F(x)$ and $G(x)$ of degree m and n, respectively. Without loss of generality we assume that $m \geq n$. Let's consider the bivariate polynomial

$$P(x,\alpha) = \frac{F(x)G(\alpha) - F(\alpha)G(x)}{x-\alpha}.$$

$P(x,\alpha)$ is a polynomial of degree $m-1$ in x and also in α. Let us represent it as

$$P(x,\alpha) = P_0(x) + P_1(x)\alpha + P_2(x)\alpha^2 + \ldots + P_{m-1}\alpha^{m-1},$$

where $P_i(x)$ is a polynomial of degree $m-1$ in x. The polynomials $P_i(x)$ can be written as follows:

$$\begin{pmatrix} P_0(x) \\ P_1(x) \\ \vdots \\ P_{m-1}(x) \end{pmatrix} = \begin{pmatrix} P_{0,0} & P_{0,1} & \ldots & P_{0,m-1} \\ P_{1,0} & P_{1,1} & \ldots & P_{1,m-1} \\ \vdots & \vdots & \vdots & \vdots \\ P_{m-1,0} & P_{m-1,1} & \ldots & P_{m-1,m-1} \end{pmatrix} \begin{pmatrix} 1 \\ x \\ x^2 \\ \vdots \\ x^{m-1} \end{pmatrix}$$

Let us denote the $m \times m$ matrix by M. The determinant of M is the resultant of $F(x)$ and $G(x)$ [Sal85]. Let us assume that $x = x_0$ is a common root of the two polynomials. Therefore, $P(x_0,\alpha) = 0$ for all α. As a result $P_i(x_0) = 0$ for $0 \leq i < m$. This condition corresponds to the fact that M is singular and $[1\ x_0\ x_0^2\ \ldots\ x_0^{m-1}]^T$ is a vector in the kernel of M.

We use Cayley's resultant formulation for implicitizing parametric curves and eliminating a variable from a pair of algebraic equations representing algebraic plane curves. More details on the properties of implicit representation, computation and accuracy are given in [MD92].

2.2. Implicitizing Parametric Curves

Given a rational Bézier curve, $\mathbf{P}(t)$:

$$\mathbf{P}(t) = \left(\frac{x(t)}{w(t)}, \frac{y(t)}{w(t)}\right) = \left(\frac{\Sigma_{i=0}^n w_i X_i B_{i,n}(t)}{\Sigma_{i=0}^n w_i B_{i,n}(t)}, \frac{\Sigma_{i=0}^n w_i Y_i B_{i,n}(t)}{\Sigma_{i=0}^n w_i B_{i,n}(t)}\right).$$

To implicitize the curve we consider the following system of equations

$$\begin{aligned} Xw(t) - Wx(t) &= 0 \\ Yw(t) - Wy(t) &= 0. \end{aligned} \qquad (1)$$

Consider them as polynomials in t and X, Y, W are treated as symbolic coefficients. The implicit representation corresponds to the resultant of (1) [Sed83].

We express the resultant as a matrix determinant. In this case the matrix has order n. Each entry of the matrix is of the form $\alpha_0 X + \alpha_1 Y + \alpha_2 W$, where $\alpha_0, \alpha_1, \alpha_2$ are scalars and functions of the control points and weights used to represent the Bézier curve. The algorithm for computing the entries of the matrix assumes that the polynomial are expressed in power basis. However, converting from Bézier to power basis can introduce numerical errors [FR87]. To circumvent this problem we perform a reparametrization. Given

$$\mathbf{P}(t) = \left(\frac{\Sigma_{i=0}^n w_i X_i B_{i,n}(t)}{\Sigma_{i=0}^n w_i B_{i,n}(t)}, \frac{\Sigma_{i=0}^n w_i Y_i B_{i,n}(t)}{\Sigma_{i=0}^n w_i B_{i,n}(t)}\right).$$

Dividing by $(1-t)^n$ and substituting $s = \frac{t}{(1-t)}$ results in

$$\overline{\mathbf{P}}(s) = \left(\frac{\Sigma_{i=0}^n w_i X_i \binom{n}{i} s^i}{\Sigma_{i=0}^n w_i \binom{n}{i} s^i}, \frac{\Sigma_{i=0}^n w_i Y_i \binom{n}{i} s^i}{\Sigma_{i=0}^n w_i \binom{n}{i} s^i}\right)$$

The rest of the algorithm proceeds by computing the implicit representation of $\overline{\mathbf{P}}(s)$ and computing a matrix formulation by Cayley's method as

$$M \begin{pmatrix} 1 \\ s \\ s^2 \\ \vdots \\ s^{m-1} \end{pmatrix} = M \begin{pmatrix} (1-t)^{m-1} \\ t(1-t)^{m-2} \\ t^2(1-t)^{m-3} \\ \vdots \\ t^{m-1} \end{pmatrix} \qquad (2)$$

The right hand side is obtained by substituting $s = \frac{t}{(1-t)}$ and multiplying vector by $(1-t)^{m-1}$. Later on, this relationship will be used to compute the inverse coordinates of the intersection points.

2.3. Intersecting Parametric Curves

Given two rational Bézier curves, $\mathbf{P}(t)$ and $\mathbf{Q}(u)$ of degree m and n respectively, the intersection algorithm proceeds by implicitizing $\mathbf{P}(t)$ and obtaining a $m \times m$ matrix M, whose entries are linear combinations of symbolic coefficients X, Y, W. The second parametrization $\mathbf{Q}(u) = (\overline{x}(u), \overline{y}(u), \overline{w}(u))$ is substituted into the matrix formulation. It results in a matrix polynomial $M(u)$ such that each of its entry is a linear combination of $\overline{x}(u), \overline{y}(u)$ and $\overline{w}(u)$. The intersection points correspond to the roots of

$$\text{Determinant}(M(u)) = 0. \qquad (3)$$

2.4. Intersecting Algebraic Curves

In this section we consider the intersection of two algebraic plane curves. They are represented as zeros of $F(x,y)$ and $G(x,y)$, polynomials of degree m and n, respectively. The polynomials may be represented in power basis or Bernstein basis. Let the points of intersection in the domain of interest be $(x_1,y_1), \ldots, (x_k,y_k)$. To simplify the problem we compute the projection of these points on the x-axis. Algebraically projection corresponds to computing the resultant of $F(x,y)$ and $G(x,y)$ by treating them as polynomials in y and expressing the coefficients as polynomials in x.

We compute the resultant using Cayley's formulation. In case, the curves are expressed in Bernstein basis, we use the reparametrization highlighted in the previous section for implicitization. The resultant is expressed as a matrix determinant and each entry of the matrix is a polynomial in x. Let us denote the matrix by $M(x)$. The problem of intersection corresponds to computing roots of Determinant$(M(x)) = 0$.

3. Matrix Computations

In this section we present techniques from linear algebra and numerical analysis. We also highlight the numerical accuracy of the problems and the algorithm used to solve these problems in terms of their *condition numbers*.

3.1. Eigenvalues and Eigenvectors

Given a $n \times n$ matrix A, its eigenvalues and eigenvectors are the solutions to the equation

$$A\mathbf{x} = \lambda \mathbf{x},$$

where λ is the eigenvalue and $\mathbf{x}$ is the eigenvector. The eigenvalues of a matrix are the roots of its characteristic polynomial determinant$(A - \lambda I) = 0$. An eigenvalue, λ_i, of multiplicity k corresponds to a root of multiplicity k of the characteristic polynomial. This multiplicity is also defined as the *algebraic multiplicity* of the eigenvalue. Moreover, the dimension of the kernel of $(A - \lambda_i I)$ is the *geometric multiplicity* of λ_i. The geometric multiplicity is bounded by the algebraic multiplicity.

Most eigendecomposition algorithms make use of the symmetric orthogonal transformations of the form $A' = QAQ^{-1}$, where Q is any non-singular orthogonal $n \times n$ matrix. This transformation has the characteristic that the eigenvalues of A and A' are identical. Furthermore, if $\mathbf{y}$ is an eigenvector of A', $Q^{-1}\mathbf{y}$ is an eigenvector of A. The running time of the eigendecomposition algorithms is $O(n^3)$. However, the constant in front of n^3 can be as high as 25 for computing all the eigenvalues and eigenvectors.

3.2. Generalized Eigenvalue Problem

Given $n \times n$ matrices, A and B, the generalized eigenvalue problem corresponds to solving

$$A\mathbf{x} = \lambda B\mathbf{x}.$$

We represent this problem as eigenvalues of $A - \lambda B$. The vectors $\mathbf{x}$ correspond to the eigenvectors of this equation. If B is non-singular and its condition number (defined in the next section) is low, the problem can be reduced to an eigenvalue problem by multiplying both sides of the equation by B^{-1} and thereby obtaining:

$$B^{-1}A\mathbf{x} = \lambda \mathbf{x}.$$

However, B may have a high condition number and such a reduction can cause numerical problems. Algorithms for the generalized eigenvalue problems apply orthogonal transformations to A and B. In particular, we use the QZ algorithm for computing the eigenvalues and eigenvectors for this problem [GL89]. Its running time is $O(n^3)$. However, the constant can be as high as 75. Generally, it is slower by a factor of 2.5 to 3 as compared to QR algorithm for computing eigenvalues and eigenvectors of a matrix.

3.3. Condition Numbers

The condition number of a problem measures the sensitivity of a solution to small changes in the input. A problem is *ill-conditioned* if its condition number is large, and *ill-posed* if its condition number is infinite. These condition numbers are used to bound errors in computed solutions of numerical problems. More details on condition numbers are given in [GL89, Wil65]. The implementations of these condition number computations are available as part of LAPACK [BDM89].

In our algorithm, we perform computations like matrix inverse and computing eigenvalues and eigenvectors of a matrix. Therefore, we are concerned with the numerical accuracy of these operations.

4. Reduction to Eigenvalue Problem

In this section we consider the problem of intersecting parametric curves and reduce it computing an eigendecomposition of a matrix. The same reduction is applicable to the intersection of algebraic plane curves as explained in [MD92].

In section 2 we had reduced the problem of intersecting parametric curves, $\mathbf{P}(t)$ and $\mathbf{Q}(u)$ of degree m and n, respectively, to finding roots of a matrix determinant as shown in (3). Each entry of the $m \times m$ matrix, $M(u)$, is a linear combination of Bernstein polynomials of degree n in u. Let us represent it as a matrix polynomial

$$M(u) = M_n u^n + M_{n-1}u^{n-1}(1-u) + \ldots + M_0(1-u)^n,$$

where M_i is a matrix with numeric entries. On dividing the equation by $(1-u)^n$ we obtain a polynomial of the form

$$M_n(\frac{u}{1-u})^n + M_{n-1}(\frac{u}{1-u})^{n-1} + \ldots + M_0.$$

Substitute $s = \frac{u}{1-u}$ and the new polynomial is of the form

$$\overline{M}(s) = M_n s^n + M_{n-1}s^{n-1} + \ldots + M_0. \quad (4)$$

In the original problem we were interested in the roots of Determinant($M(u)$) = 0 in the range $[0,1]$. However, after reparametrizing we want to compute the roots of Determinant($\overline{M}(s)$) = 0 in the range $[0,\infty]$. This can result in overflow problems if the original system has a root $u \approx 1$. In such cases M_n is nearly singular or ill–conditioned. Our algorithm takes care of such cases by linear transformations or using projective coordinates.

Let us consider the case when M_n is a well–conditioned matrix. Given $\overline{M}(s)$, we multiply the matrix polynomial by M_n^{-1}. As a result we obtain

$$\overline{M}'(s) = I_m s^n + \overline{M}_{n-1}s^{n-1} + \ldots + \overline{M}_0,$$

where I_m is an $m \times m$ identity matrix and $\overline{M}_i = M_n^{-1}M_i$ for all $i < n$. Given the matrix polynomial $\overline{M}'(s)$ we compute a $nm \times nm$ matrix of the form

$$C = \begin{pmatrix} 0 & I_m & 0 & \ldots & 0 \\ 0 & 0 & I_m & \ldots & 0 \\ \vdots & \vdots & \vdots & \vdots & \vdots \\ 0 & 0 & 0 & \ldots & I_m \\ -\overline{M}_0 & -\overline{M}_1 & -\overline{M}_2 & \ldots & -\overline{M}_{n-1} \end{pmatrix}, \quad (5)$$

such that the eigenvalues of C correspond exactly to the roots of Det($\overline{M}(s)$) = 0. Furthermore C is a numeric matrix of order mn. The proof of this property is given as part of Theorem 1.1 [GLR82]. The relationship between C and $\overline{M}'(s)$ is identical to that between the characteristic polynomial of a matrix and its equivalent companion matrix [Wil65].

Let λ be an eigenvalue of C. As a result $\overline{M}'(\lambda)$ is a singular matrix. Let $\mathbf{v} = [v_1\ v_2\ \ldots\ v_m]^T$ be the vector in the kernel of $\overline{M}'(\lambda)$ such that

$$\overline{M}'(\lambda)(v_1\ v_2\ \ldots\ v_m)^T \;=\; (0\ 0\ \ldots\ 0)^T. \quad (6)$$

Corollary I: *The eigenvector of C corresponding to the eigenvalue* λ *has the form* $\mathbf{V} = [\mathbf{v}\ \lambda\mathbf{v}\ \lambda^2\mathbf{v}\ \ldots\ \lambda^{n-1}\mathbf{v}]^T$, *where* $\mathbf{V}$ *is a* $mn \times 1$ *vector.*
Proof: [MD92].

It follows that the eigenvalues of C correspond exactly to the preimages of intersection points on $\mathbf{Q}(u)$. However, we are only interested in the eigenvalues in the range $s_0 \in [0,\infty]$ and the preimages on the curve are obtained by substituting $u_0 = \frac{s_0}{1+s_0}$. This gives us a list of all the intersection points on $\mathbf{Q}(u_0)$ such that $u_0 \in [0,1]$. However, these points on $\mathbf{P}(t)$ may not lie in the range $t \in [0,1]$. As a result it is important for us to compute the preimage of the intersection point $(x_0, y_0, w_0) = \mathbf{Q}(u_0)$ with respect to $\mathbf{P}(t)$. We use the property of the linear system of equations (2) and Corollary I.

Let us assume that (x_0, y_0, w_0) is a simple point on $\mathbf{P}(t)$. Points of higher multiplicity are accounted for in the next section. Substitute for $(X, Y, W) = (x_0, y_0, w_0)$ in the matrix, M as shown in (2), corresponding to the implicit representation of $P(t)$. The resulting matrix is singular and let us assume that its kernel has dimension one. Kernels of higher dimension correspond to higher order intersection. The vector in the kernel corresponds to $\mathbf{v}$ shown in (6). Given the eigenvector of C corresponding to the eigenvalue s_0, use Corollary I to compute the eigenvector $\mathbf{v}$. Given $\mathbf{v}$ we use the structure of the linear system to compute the preimage of the point (x_0, y_0, w_0) by using the relation

$$\left((1-t_0)^{m-1}\ t_0(1-t_0)^{m-2}\ \ldots\ t_0^{m-1}\right)^T = k\,(v_1\ v_2\ \ldots\ v_m)^T,$$

where $k \neq 0$ is a constant. Thus, $t_0 = \frac{v_2}{v_1+v_2}$. The relationship between the eigenvalue s_0 of C, elements v_1, v_2 of the eigenvector $\mathbf{V}$ corresponding to s_0 and the point of intersection (x_0, y_0, w_0) can be expressed as

$$(x_0, y_0, w_0) = \mathbf{Q}(\frac{s_0}{1+s_0}) = \mathbf{P}(\frac{v_2}{v_1+v_2}). \quad (7)$$

As a result we are able to compute all the points of intersection in the domain of interest by computing the eigendecomposition of C.

Let us consider the case, when the matrix M_n in (4) is ill–conditioned. One such case occurs when the preimage of the point of intersection on $\mathbf{P}(t)$ is $t_0 \approx 1$. As a result, computation of M_n^{-1} and the corresponding reduction to the eigenvalue problem can introduce numerical problems. The general approach for solving such cases is the reduction to generalized eigenvalue problem. In this case we construct companion matrices of the form [GLR82, Section 7.2]

$$C_1 = \begin{pmatrix} I_n & \ldots & 0 & 0 \\ 0 & I_n & \ldots & 0 \\ \vdots & \vdots & \vdots & \vdots \\ 0 & \ldots & I_n & 0 \\ 0 & \ldots & 0 & M_n \end{pmatrix}, C_2 = \begin{pmatrix} 0 & -I_n & 0 & \ldots & 0 \\ 0 & 0 & -I_n & \ldots & 0 \\ \vdots & \vdots & \vdots & \vdots & \vdots \\ 0 & 0 & \ldots & 0 & -I_n \\ M_0 & M_1 & \ldots & M_{n-2} & M_{n-1} \end{pmatrix}$$

such that the eigenvalues of $C_1 s + C_2$ correspond exactly to the roots of Det($M(s)$) = 0. Furthermore, the eigenvectors of this generalized system have a structure similar to $\mathbf{V}$ highlighted in Corollary I and is used for computing the inverse image of the intersection point with respect to $\mathbf{P}(t)$.

Solving a generalized eigenvalue system is more expensive than the normal eigenvalue system (almost a factor of 3). In many cases, we can perform a linear transformation on the coordinate of the matrix polynomial and reduce the resulting problem to an eigenvalue problem. The basic idea involves transforming $s = \frac{a\overline{s}+b}{c\overline{s}+d}$, where a, b, c and d are random numbers. The matrix polynomial $\overline{M}(s)$ in (4) is transformed into

$$P(\overline{s}) = (c\overline{s} + d)^n M(\frac{a\overline{s}+b}{c\overline{s}+d})$$

$$\Rightarrow P(\bar{s}) = P_n\bar{s}^n + P_{n-1}\bar{s}^{n-1} + \ldots + P_1\bar{s} + P_0,$$

where P_i's are computed from the M_j's. If P_n is a well-conditioned matrix then the problem of intersection is reduced to an eigenvalue problem, otherwise use a different transformation (by a different choice of a,b,c and d). The linear transformation is performed up to four or five times. If all the resulting leading matrices, P_n, are ill-conditioned, the intersection problem is reduced to a generalized eigenvalue problem. There are cases when any linear transformation can result in an ill-conditioned leading matrix. Furthermore, the domain of the eigenvalue system obtained after transformation is $[s_1, s_2]$ or $[s_2, s_1]$ depending upon the signs of a, b, c and d, where $s_1 = -\frac{b}{a}$ and $s_2 = -\frac{d}{c}$.

4.1. Implementation and Performance

The reduction to an eigenvalue or a generalized eigenvalue system involves estimating the condition number of a matrix, computing the matrix inverse and finding the eigenvalues of a matrix. For eigenvalues lying in the domain of interest, we compute the corresponding eigenvectors. Furthermore, we also compute the condition number of each eigenvalue in the domain of interest. The condition number is a function of the left and right eigenvectors of the matrix.

We used LAPACK implementation of eigendecomposition algorithms. Some of the routines were modified to compute the eigenvalues in the domain of interest. Furthermore, the domain was specified as $\alpha + j\beta$, where $\alpha > -\epsilon$, $|\beta| < \epsilon$ and $j = \sqrt{-1}$. ϵ is a small positive constant used to account for the numerical errors. In particular, we make ϵ a function of the condition number of M_n or P_n for eigenvalue problems. To compute the inverse coordinate of the intersection point, the right eigenvector $\mathbf{V}$ corresponding to the eigenvalue s_0 is computed. Let

$$\mathbf{V} = [v_{1,1}\ v_{1,2}\ \ldots\ v_{1,m}\ v_{2,1}\ \ldots\ v_{2,m}\ \ldots\ v_{n,1}\ \ldots\ v_{n,m}]^T.$$

Analysis of the accuracy of eigenvector computation indicates that each term of the eigenvector has a similar bound on the absolute error of the computation. As a result we tend to use terms of maximum magnitude to minimize the error in the computation [MD92]. In this case we compute the entries of $\mathbf{v} = [v_1\ v_2\ \ldots\ v_m]^T$ as:

$$[v_1\ v_2\ \ldots\ v_m]^T = \begin{cases} [v_{1,1}\ v_{1,2}\ \ldots\ v_{1,m}]^T & s_0 \le 1 \\ \frac{1}{(s_0)^n}[v_{n,1}\ v_{n,2}\ \ldots\ v_{n,m}]^T & s_0 > 1 \end{cases}$$

Given $\mathbf{v}$ the inverse coordinate, t_0, is computed using v_1, v_2 or v_{m-1}, v_m by making use of similar numerical properties.

The performance of the algorithm is largely governed by the eigendecomposition routines. Roughly 80 – 85% of the time is spent in these routines. The eigenvalue algorithms compute all the eigenvalues of the given matrix. It is difficult to restrict them to computing eigenvalues in the domain of interest. The order of the matrix, say p, corresponds to the product of the degree of the two curves and the number of eigenvalues is equal to the order. The running time of the algorithm is a cubic function of p. The eigenvalue algorithms are iterative and have good convergence properties. It is a long observed fact that the algorithm requires two iterations per eigenvalue. As a result it is possible to bound the actual running time of the eigenvalue computation by $10p^3$ for most cases. Furthermore the eigendecomposition algorithms are backward stable. We have been to able accurately compute the intersections of curves of degree up to ten. In practice it is possible to obtain accurate solutions for matrices of order 100 or more. This is in contrast with computing roots of high degree univariate polynomials (which is an ill-conditioned problems) or using symbolic computation for determinant computation and finding the roots of the resulting polynomial expressed in Bernstein basis using subdivision and iteration (which is relatively expensive and has slow convergence).

4.2. Example of Curve Intersection

In this section we illustrate the algorithm by considering the intersection of two rational cubic Bézier curves. The example is taken from [Sed83]. The control points of two Bézier curves (as shown in Fig. II), expressed in homogeneous coordinates, are $(4,1,1), (5,6,2), (5,0,2), (6,4,1)$ and $(7,4,1), (1,2,2), (9,2,2), (3,4,1)$. Thus,

$$\mathbf{P}(t) = \left(\frac{x(t)}{w(t)}, \frac{y(t)}{w(t)}\right),$$

where

$$x(t) = 4(1-t)^3 + 30(1-t)^2t + 30(1-t)t^2 + 6t^3$$

$$y(t) = (1-t)^3 + 36(1-t)^2t + 4t^3$$

$$w(t) = (1-t)^3 + 6(1-t)^2t + 6(1-t)t^2 + t^3.$$

The implicit representation has a matrix determinant formulation given as

$$M = \begin{pmatrix} -114w + 30x - 6y & 30w - 6x - 6y & -10w + 3x - 2y \\ 30w - 6x - 6y & 1070w - 213x - 2y & 96w - 12x - 6y \\ -10w + 3x - 2y & 96w - 12x - 6y & -120w + 24x - 6y \end{pmatrix}.$$

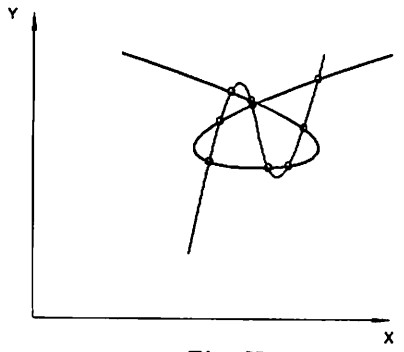


Fig. II
Intersection of rational cubic Bézier curves

The second parametrization, $\mathbf{Q}(u)$ is substituted into the matrix formulation (after a reparametrization of the form $s = \frac{u}{1-u}$). The resulting matrix polynomial has the form

$$\overline{M}(s) = \begin{pmatrix} -48 & -12 & -9 \\ -12 & 423 & 36 \\ -9 & 36 & -72 \end{pmatrix} s^3 + \begin{pmatrix} 864 & -216 & 78 \\ -216 & -5106 & -144 \\ 78 & -144 & 504 \end{pmatrix} s^2$$
$$+ \begin{pmatrix} -576 & 72 & -66 \\ 72 & 5118 & 432 \\ -66 & 432 & -648 \end{pmatrix} s + \begin{pmatrix} 72 & -36 & 3 \\ -36 & -429 & -12 \\ 3 & -12 & 24 \end{pmatrix}.$$

The condition number of the leading matrix is 9.525.

Multiplying $M(s)$ with the inverse of the leading matrix and constructing the equivalent companion matrix results in

$$C = \begin{pmatrix} 0 & 0 & 0 & 1 & 0 & 0 & 0 & 0 & 0 \\ 0 & 0 & 0 & 0 & 1 & 0 & 0 & 0 & 0 \\ 0 & 0 & 0 & 0 & 0 & 1 & 0 & 0 & 0 \\ 0 & 0 & 0 & 0 & 0 & 0 & 1 & 0 & 0 \\ 0 & 0 & 0 & 0 & 0 & 0 & 0 & 1 & 0 \\ 0 & 0 & 0 & 0 & 0 & 0 & 0 & 0 & 1 \\ 1.48 & -1.07 & 0 & -11.92 & 4.58 & 0.40 & 17.8 & -8.24 & 0.40 \\ 0.13 & 0.94 & 0 & -0.53 & -11.92 & -.22 & 1.06 & 11.4 & -0.23 \\ -.07 & .44 & .33 & .30 & -.53 & -9.16 & -.61 & 4.74 & 6.83 \end{pmatrix}.$$

The eigendecomposition of C results in 9 points of intersection as shown in the following table.

Intersection Number	Eigenvalue s_0	Max. Error E_i	Parameter $u_0 = \frac{s_0}{1+s_0}$	Eigenvector's $\alpha = v_1 \vert v_8$	Eigenvector's $\beta = v_2 \vert v_9$	Parameter $t_0 = \frac{\beta}{\alpha+\beta}$	Point (X,Y)
1.	15.369	2.32e-14	0.9389	0.2173	0.0472	0.1785	$(4.619, 3.412)$
2.	11.802	2.85e-14	0.9219	0.6657	0.4432	0.3997	$(4.911, 3.289)$
3.	5.507	2.71e-14	0.8463	0.0703	1.000	0.9343	$(5.688, 2.877)$
4.	1.4654	1.27e-13	0.5944	0.1614	1.000	0.8610	$(5.467, 2.321)$
5.	0.5361	2.32e-14	0.3490	1.00	0.066	0.0622	$(4.298, 2.378)$
6.	0.1534	2.98e-14	0.133	1.00	0.1233	0.1099	$(4.455, 2.971)$
7.	0.0974	2.38e-14	0.0888	1.00	0.7277	0.4212	$(4.931, 3.218)$
8.	1.145	1.18e-13	0.534	1.00	0.4644	0.683	$(4.174, 2.290)$
9.	0.0382	1.14e-14	0.0369	0179	0.00032	0.9823	$(5.901, 3.615)$

Eigendecomposition and intersection points

The intersections points are computed using the relationship highlighted in (7). They are: In the columns corresponding to the components of the eigenvectors we choose the elements v_1 or v_8 depending upon their relative magnitudes. The error bounds in the third column are obtained by using the condition number of the eigenvalues (as explained in [MD92]) and matrix norm as

$$E_i = \epsilon \parallel C \parallel \mathrm{cond}_i, \qquad (8)$$

where $\epsilon = 2.2204e-16$ is the machine precision for 64 bit IEEE floating point arithmetic and cond_i is the condition number of the ith eigenvalue. As a result, the eigendecomposition algorithms compute the eigenvalues of C up to 12 digits of accuracy. The other sources of error arise from the computation of the entries of M, the matrix corresponding to the implicit representation, and inverting the leading matrix of the matrix polynomial $\overline{M}(s)$. In our case, this can account for inaccuracy of one digit (due to condition number of the matrix to be inverted). As a result, the intersection points are computed up to 11 digits of accuracy. Further accuracy can be obtained by using one or two iterations of Newton's method on the equations used for representing the problem of intersection. The output of the algorithm are the starting points for Newton's method. As a result the algorithm can be used to compute intersection points with high accuracy.

5. Higher Order Intersection

In the previous section, we presented an algorithm to compute the simple intersections of parametric curves. In this section we extend the analysis to higher order intersections.

According to Bezout's theorem two rational curves of degree m and n intersect in mn points (counted properly) [Wal50]. In our case, the preimages of these points correspond to the mn eigenvalues of C (5). Higher order intersections are the points having more than one preimage. In other words, eigenvalues of multiplicity greater than one correspond to higher order intersection points. The *intersection multiplicity* of these points corresponds to the algebraic multiplicity of the corresponding eigenvalue. These intersections arise due to the tangential intersection of the two curves at the point of contact or the intersection point is a singular point on at least one of the curves. Some higher order intersections are highlighted in Fig. III. They are:

(a) Tangential intersection of two ellipses. The intersection multiplicity is 2.

(b) Second order intersection of a parabola with a curve having a loop.

(c) Intersection of an ellipse and a curve with a cusp. The intersection multiplicity is 2.

(d) Tangential intersection of a parabola with a loop. The intersection multiplicity is 3.

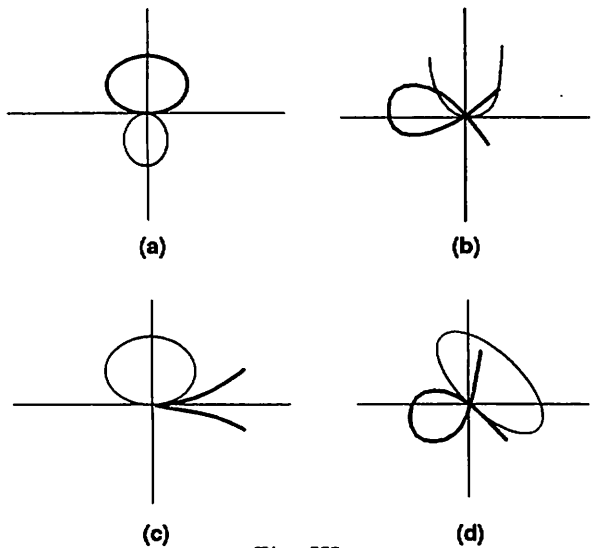


Fig. III
Higher order intersections

An intersection point, P, has multiplicity k, if a small perturbation in the coefficients of the curve (or the coefficients of the control polygon) results in k distinct intersection points in the neighborhood of P. Given two curves that intersect with multiplicity k at P, our algorithm computes the implicit representation and reduces the problem to an eigenvalue problem. The latter computation involves matrix inversion and multiplications using floating point operations. As a result, C in (5), corresponds to a slightly perturbed problem and k of its eigenvalues, say $\lambda_1, \lambda_2, \ldots, \lambda_k$ are very close to λ, the eigenvalue corresponding to P (assuming exact arithmetic). The eigendecomposition algorithms use floating point arithmetic and the computed eigenvalues correspond to $\lambda_1', \lambda_2', \ldots, \lambda_k'$. However the problem of computing eigenvalues of multiplicity greater than one can be ill-conditioned [GL89, Wil65]. As a result λ_i' may agree with λ up to a few digits of accuracy. This can be determined from the condition number of λ_i'. In such cases the mean of λ_i''s given by

$$\lambda' = \frac{\lambda_1' + \lambda_2' + \ldots + \lambda_k'}{k}$$

is much better conditioned. This can be verified by computing the condition number of a cluster of eigenvalues [BDM89]. As a result λ' is very close to λ, the eigenvalue of multiplicity k. The number k corresponds to the number of ill-conditioned eigenvalues, λ_i', located in the small neighborhood of each other. We illustrate this technique using the following example.

Example: Consider the intersection of cubic curve, $\mathbf{P}(t) = (x(t), y(t)) = (t^2 - 1, t^3 - t)$ with the parabola $\mathbf{Q}(u) = (\overline{x}(u), \overline{y}(u)) = (u^2 + u, u^2 - u)$ (as shown in Fig. IV). The cubic curve has a loop at the origin. We are interesting in computing all the intersection points corresponding to the domain $t \times u = [-2, 2] \times [-1, 1]$.

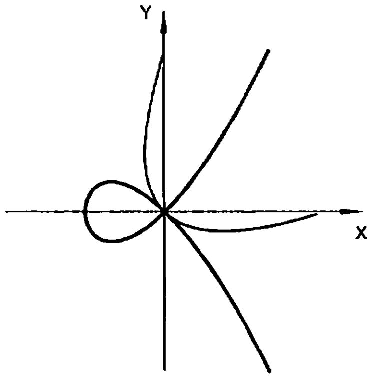


Fig. V
Higher order intersection of a cubic curve and a parabola

The implicit representation of $\mathbf{P}(t)$ is a matrix determinant of the form

$$M = \begin{pmatrix} -w - x & -y & w + x \\ -y & x & 0 \\ w + x & 0 & -w \end{pmatrix}.$$

After substituting and reducing the problem to an eigenvalue problem we obtain a 6×6 matrix

$$C = \begin{pmatrix} 0 & 0 & 0 & 1 & 0 & 0 \\ 0 & 0 & 0 & 0 & 1 & 0 \\ 0 & 0 & 0 & 0 & 0 & 1 \\ -1 & 0 & 1 & -1 & 0 & 0 \\ -1 & 0 & 1 & -2 & -1 & 0 \\ -1 & 0 & 1 & -2 & -2 & -1 \end{pmatrix}.$$

The relevant eigenvalues of this matrix and their condition numbers are:

Intersection Number	Eigenvalue λ_i'	Cond. Number Cond_i
1.	0.0	1.2171
2.	-0.00000001296346	4.86e09
3.	0.00000001296346	4.86e09

Eigenvalues corresponding to higher order intersections

Thus, the second and third eigenvalues have a high condition number. Taking their average we obtain $\lambda' = 0.0$, and it has a low condition number. As a result, we conclude that $\lambda' = 0.0$ is an eigenvalue of multiplicity 3 and the curves have a triple intersection at $\mathbf{Q}(0.0) = (0.0, 0.0)$.

Q.E.D.

The procedure highlighted above is used for computing the intersection multiplicity of the point. However, the intersection can arise from tangential intersection, singular points or their combinations. The rest of the analysis deals with determining the nature of intersection.

Given an eigenvalue, λ, of algebraic multiplicity k, its geometric multiplicity corresponds to the dimension of the kernel of $(C - \lambda I)$. After accurately computing the algebraic multiplicity of the eigenvalue, we compute its geometric multiplicity. Furthermore, the algorithm computes a basis of the vectors spanning the kernel. Depending upon the nature of intersection the geometric multiplicity is less than or equal to algebraic multiplicity. The exact relationship between the multiplicities and nature of intersection has been described in [MD92]. Here we highlight the relationship for the examples in Fig. III and IV. We assume that the curves drawn in dark font are being implicitized. The curves highlighted in light font are substituted into the implicit representation. The choice of curve for implicitization has an impact on the geometric multiplicities of the matrix, although the algebraic multiplicities are unaffected [MD92].

Example Figure	Algebraic Multiplicity	Geometric Multiplicity
III(a)	2	1
III(b)	2	2
III(c)	2	2
III(d)	3	2
IV	3	2

Algebraic and geometric multiplicities of eigenvalues corresponding to Figs. III and IV

In case the geometric multiplicity is 1, the computation of the preimage corresponds exactly to the procedure described in the previous section. If the geometric multiplicity is greater than 1, the algorithm for preimage computation involves equation solving. Let us illustrate for Fig. V.

In the example corresponding to Fig. V, the intersection multiplicity is 3. If we implicitize the parabola and substitute the cubic curve into the implicit form, the eigenvalue corresponding to the origin has algebraic multiplicity 3 and geometric multiplicity 1. This is due to the fact that the parabola is intersecting the cubic curve tangentially at the loop, corresponding to the origin. The fact that the intersection point is a loop implies that $\mathbf{P}(t)$ has two distinct preimages $t_1 = 1$ and $t_2 = -1$. As a result both the vectors $\mathbf{v}_1 = [1\, t_1\, t_1^2]^T$ and $\mathbf{v}_2 = [1\, t_2\, t_2^2]^T$ lie in the kernel of M after we substitute $x = 0, y = 0, w = 1$. Since these vectors are linearly independent they constitute the basis of the eigenvectors corresponding to $\lambda = 0$. However, the eigendecomposition algorithm can return any two linearly independent vectors of the form $\mathbf{V}_1 = \alpha_1\mathbf{v}_1 + \alpha_2\mathbf{v}_2$ and $\mathbf{V}_2 = \beta_1\mathbf{v}_1 + \beta_2\mathbf{v}_2$, where α_i's and β_j's are scalars. The rest of the algorithm involves computing $\mathbf{v}_1$ and $\mathbf{v}_2$ from $\mathbf{V}_1$ and $\mathbf{V}_2$. This corresponds to computing the scalars and can be achieved by equation solving [MD92].

6. Conclusion

In this paper we have highlighted a new technique for computing the intersection of parametric and algebraic curves. The algorithm involves use of resultants to represent the implicit representation of a parametric curve as a matrix determinant. The intersection problem is being reduced to an eigenvalue problem. The algorithm is very robust and can accurately compute the intersection points. There is a nice relationship between the algebraic and geometric multiplicities of the eigenvalues and the order of intersection. We used this relationship in accurately computing such intersections. Efficient implementations of eigenvalue routines are available as part of linear algebra packages and we used them in our implementations.

The approach highlighted here is also useful for intersecting curves and surfaces. In particular, the implicit representation of a parametric surface can be represented as a matrix determinant [MC91]. The parametric space curve is substituted into the implicit formulation and the problem can therefore, be reduced to an eigenvalue problem. This can be directly used for *ray tracing* parametric patches, as a ray corresponds to degree one parametric curve.

Acknowledgement: Dinesh Manocha is supported in part by IBM Graduate Fellowship, David and Lucile Packard Fellowship and National Science Foundation grant ASC-9005933. James Demmel is supported in part by National Science Foundation grants DCR-8552474, ASC-8715728 and ASC-9005933.

References

[ABB+90] E. Anderson, Z. Bai, C. Bischof, J. Demmel, J. Dongarra, J. Du Croz, A. Greenbaum, S. Hammarling, and D. Sorensen. Lapack: A portable linear algebra library. Computer Science Technical Report CS-90-105, University of Tennessee, 1990.

[BBB87] R.H. Bartels, J.C. Beatty, and B.A. Barsky. *An Introduction to Splines for use in Computer Graphics & Geometric Modeling.* Morgan Kaufmann, San Mateo, California, 1987.

[BDM89] Z. Bai, J. Demmel, and A. McKenney. On the conditioning of the nonsymmetric eigenproblem: Theory and software. Computer Science Dept. Technical Report 469, Courant Institute, New York, NY, October 1989. (LAPACK Working Note #13).

[EC90] G. Elber and E. Cohen. Hidden curve removal for free form surfaces. *Computer Graphics*, 24(4):95–104, 1990.

[FR87] R.T. Farouki and V.T. Rajan. On the numerical condition of polynomials in Bernstein form. *Computer Aided Geometric Design*, 4:191–216, 1987.

[GL89] G.H. Golub and C.F. Van Loan. *Matrix Computations.* John Hopkins Press, Baltimore, 1989.

[GLR82] I. Gohberg, P. Lancaster, and L. Rodman. *Matrix Polynomials.* Academic Press, New York, 1982.

[Hof89] C.M. Hoffmann. *Geometric and Solid Modeling.* Morgan Kaufmann, San Mateo, California, 1989.

[KM83] P.A. Koparkar and S. P. Mudur. A new class of algorithms for the processing of parametric curves. *Computer-Aided Design*, 15(1):41–45, 1983.

[LR80] J.M. Lane and R.F. Riesenfeld. A theoretical development for the computer generation and display of piecewise polynomial surfaces. *IEEE Transactions on Pattern Analysis and Machine Intelligence*, 2(1):150–159, 1980.

[MC91] D. Manocha and J.F. Canny. A new approach for surface intersection. In *First ACM Symposium on Solid Modeling Foundations and CAD/CAM Applications*, pages 209–220, 1991. Revised version to appear in *International Journal of Computational Geometry and Applications.*

[MD92] D. Manocha and J. Demmel. Algorithms for intersecting parametric and algebraic curves. Technical report, Computer Science Division, UC Berkeley, 1992.

[MM89] R.P. Markot and R. L. Magedson. Solutions of tangential surface and curve intersections. *Computer-Aided Design*, 21(7):421–427, 1989.

[Sal85] G. Salmon. *Lessons Introductory to the Modern Higher Algebra.* G.E. Stechert & Co., New York, 1885.

[Sed83] T.W. Sederberg. *Implicit and Parametric Curves and Surfaces.* PhD thesis, Purdue University, 1983.

[Sed89] T.W. Sederberg. Algorithms for algebraic curve intersection. *Computer-Aided Design*, 21(9):547–555, 1989.

[SP86] T.W. Sederberg and S.R. Parry. Comparison of three curve intersection algorithms. *Computer-Aided Design*, 18(1):58–63, 1986.

[SWZ89] T.W. Sederberg, S. White, and A. Zundel. Fat arcs: A bounding region with cubic convergence. *Computer Aided Geometric Design*, 6:205–218, 1989.

[Wal50] R.J. Walker. *Algebraic Curves.* Princeton University Press, New Jersey, 1950.

[Wil59] J.H. Wilkinson. The evaluation of the zeros of ill-conditioned polynomials. parts i and ii. *Numer. Math.*, 1:150–166 and 167–180, 1959.

[Wil65] J.H. Wilkinson. *The algebraic eigenvalue problem.* Oxford University Press, Oxford, 1965.

An Interval Refinement Technique for Surface Intersection

Michael Gleicher
School of Computer Science
Carnegie Mellon University
Pittsburgh, PA 15213-3890
gleicher@cs.cmu.edu

Michael Kass
Apple Computer
20525 Mariani Ave.
Cupertino, CA 95014
kass@apple.com

Abstract

This paper describes a technique for computing the intersections of two parametric surfaces based on interval arithmetic. The algorithm, which can be stopped and restarted at any point, uses search techniques to refine its description of the intersections progressively. Interval arithmetic provides guaranteed points on the intersection curves to within a user-specified tolerance. These points are connected into polygons and used to triangulate the trimmed surfaces. We provide details of an implementation and give examples of the algorithm's use.

Résumé

Cet article décrit une technique pour calculer les intersections de deux surface paramétriques avec l'intervalle arithmétique. L'algorithme utilise des techniques de recherche pour augmenter la précision des intersections, et peut être interrompu ou redemarrer à n'importe quel moment. L'intervalle arithmétique nous donne des points garantis sur les courbes d'intersections. Nous créons des polygons avec les points et triangulons les regions interieurs. Nous exposon en detail un modèle de mise en oeuvre et nous donnons des exemples de l'utilisation de l'algorithme.

Keywords: Surface Intersection, Interval Arithmetic

1 Introduction

Finding the intersection of two parametric surfaces is an important problem in Computer Aided Geometric Design. It is useful in many applications such as trimming surfaces and performing boolean operations on boundary representation geometric models[17]. The difficulty of this problem forces solutions to trade generality, robustness, and performance.

Here, we present a method for intersecting parametric surfaces based on interval arithmetic. The method is very general, placing few restriction on the class of surfaces it can handle. Nonetheless, the intersection points it finds are are guaranteed to be within a specified tolerance. Since the method continually refines its results, a valid estimate of the intersection is always available during execution. As a consequence, the algorithm can be stopped when tolerance criteria or time bounds are met and restarted if the results are unacceptable. Adjusting the tolerances makes it posible to trade accuracy or sampling rate for computation time.

In sections two and three, we briefly review previous work on the intersection problem and describe the basics of interval arithmetic. Then section four describes the algorithm for finding intersection points, formulating the the task as a search problem. Finally, section five addresses the issue of linking the intersection points together and triangulating the bounded regions of parameter space. Results from our prototype implementation are presented for a variety of shapes.

2 Related Work

Surface intersection problems have been widely studied because of their practical importance (see [16] or [12] for a survey). The general problem is to find the set of points where two surfaces coincide in space. While two surfaces typically intersect at a set of space curves, the intersection may also contain distinct points or surface elements in degenerate cases.

Parametric representations define surfaces by maps from the plane to three-dimensional Euclidean space. These representations are extremely popular because of their convenience for a variety of modeling and rendering purposes. Unfortunately, parametric surfaces are very difficult to intersect[16].

In general, exact analytical solutions for surface intersection problems are unavailable or impractical, since even simple surfaces can meet at very complicated curves[11]. As a consequence practical solutions to the intersection problems must resort to approximating the solution. Following Barnhill et. al.[4], we characterize these approximations by tolerances which specify how closely the approximation must match the actual intersection.

The literature on surface intersections contains a wide variety of approaches. The most common are marching and subdivision. Marching methods (e.g. [3, 14, 12, 5, 4, 6]), begin with points known to be at the intersection of the two surfaces and use numerical techniques to compute successive points on the intersection curve. In addition to the numerical challenges of progressing around the curve, these techniques face the additional task of finding the initial points from which to begin marching.

Subdivision is another approach to finding the intersection of two surfaces. The basic idea (e.g. [16, 9, 7]) is to divide the surface intersection problem into smaller pieces until each piece is a solvable problem. For example, Houghten et. al. [9] subdivide surfaces until each piece is nearly planar and use the fact that intersections of planar elements can be calculated directly.

One difficulty with subdivision approaches is that they require a way of deciding whether the subproblems are adequately modeled by the solvable problems, for example deciding if a surface segment is nearly planar [9]. A second difficulty is that the results are often not guaranteed at all, or only guaranteed for a very restricted class of surfaces. For example, some powerful recent results (e.g. [19, 18]) apply only to polynomial or rational functions. Even if the results can be guaranteed, many subdivision techniques have poor performance [19] because they need to search exhaustively for intersections.

Our approach has several advantages over typical subdivision approaches. The use of interval arithmetic permits us to make guarantees about finding points on the intersection curve without placing severe restrictions on the class of surfaces the algorithm can handle. The parametric mappings can be expressed in terms of trigonometric functions, for example, with no particular difficulty. In addition, the subdivision strategy we use avoids exhaustive search in most cases. By adjusting the tolerances used for stopping conditions, the tradeoff between time and accuracy can be user controlled.

Interval arithmetic has been used for a variety of purposes in computer graphics[15, 13, 1, 20, 21]. Mudar and Koparkar [15] present the basic idea of using interval arithmetic to identify surface intersections but make no mention of the issues involved in creating efficient reliable algorithms which provide descriptions of intersection curves within user specified tolerances. The work of Von Herzen and Barr[22] is very similar, using Lipschitz conditions to evaluate bounding regions for portions of surfaces. The Lipschitz conditions are derived by hand for each new analytic surface, unlike the automatic interval arithmetic used here. Von Herzen and Barr also do not address the issue of finding the intersections themselves, instead relying on implicit functions for breaking objects into pieces.

3 Interval Arithmetic

Interval arithmetic is a method for providing a bound on the output of a function given bounds on all of its inputs. This section provides a brief introduction to interval arithmetic and its relevance to the surface/surface intersection problem.

Interval arithmetic is based on the idea of extending ordinary scalar operations to intervals on the real line. If S is an interval, we can write it as (S_{min}, S_{max}) to denote a quantity whose value lies somewhere between S_{min} and S_{max}. With every ordinary scalar function, say $f(s,t)$, we associate an interval function $F(S,T)$, which provides a bound on $f(s,t)$ given bounds on s and t. We begin by defining the interval functions corresponding to primitive operations (e.g. basic arithmetic operations and trigonometric functions). For example, if $f(s,t) = s + t$, we can define $F(S,T)$ to be the interval $(S_{min} + T_{min}, S_{max} + T_{max})$. Clearly, if s and t are within their bounds, their sum must lie in the interval $F(S,T)$. Similar rules can be developed for a wide variety of elementary functions[2].

Once we have defined a set of interval functions corresponding to primitive operations, we can create more complicated interval functions by composing them. The interval function corresponding to $f(g(q,r), h(s,t))$, for example, is simply $F(G(Q,R), H(S,T))$. We have automated this operation by defining a set of operations on intervals using the operator overloading capabilities of C++.

Parametric surfaces are defined by mappings from (u,v) to (x,y,z). If we use interval arithmetic to represent the mapping, then we have an interval function which maps from (U,V) to (X,Y,Z). The interval mapping provides an axis-aligned bounding box in world space for every rectangular region of parameter space. The bounding box may not be a tight bound, but we are guaranteed that it contains the piece of the parametric surface defined by the rectangular region of parameter space.

4 An Interval Approach to Surface Intersection

Suppose that we have two parametric surfaces and their corresponding interval functions. If we pick a rectangle in each parameter space, the interval functions provide a pair of bounding boxes, one for each surface. We make use of the bounding boxes as follows. If the bounding boxes do not overlap, we know that the surfaces do not intersect in the corresponding parameter-space rectangles. In that case, we need not examine these regions of parameter space any further. If the bounding boxes do overlap, the surfaces might intersect in the corresponding regions of parameter space, but we cannot be sure that they do. To learn more, we can subdivide the parameter space. Suppose we subdivide until the bounding boxes all have diagonals smaller than $\epsilon/2$. Then if we find a pair of intersecting bounding boxes from the two surfaces, we can conclude that the surfaces approach each other to within a distance of ϵ in the corresponding regions of parameter space. We refer the corresponding parameter-space regions in such a case as a "dot" and its "mate." Each dot gives us a point on the intersection curve to within the dot tolerance ϵ. The problem is to find an appropriate set of dots which can be linked together to form the trimming curve.

4.1 A Simple Interval Intersection Method

One way to find an appropriate set of dots is to divide each parameter space into a uniform grid, but the cost of such a subdivision is prohibitive. Instead, interval algorithms usually divide space hierarchically, only dividing up regions of space which may contain solutions. In addition to time and space efficiency, the hierarchical alogithms have the advantage of progressive refinement approaches: at all times there is a valid approximation of the entire solution and the approximation improves as the algorithm progresses.

To avoid having to compare every square in one parameter space against every square in the other, each leaf node of the tree maintains a list of the leaf nodes in the other tree which it overlaps. Since the bounding volume of a child must be completely contained within the volume of its parent, when we subdivide, we only need to check the new children against the boxes intersected by their parents. The subdivision step of our algorithm is:

```
Subdivide(node)
  if node's intersect list is not empty
    subdivide node into children
    for each i in node's intersect list
        remove node from i's intersect list
        for each child of node
            if child overlaps i
              add i to child's intersect list
              add child to i's intersect list
```

The basic algorithm for finding intersections is to pick a leaf node from one of the trees and subdivide it. A list of "live" leaf nodes (i.e.: ones which contain overlaps) provides a description of the current model of the intersection curve as well as a "to do" queue. How the next node to be divided is chosen from the list of potential choices provides control over the search algorithm. An obvious choice is searching breadth-first by always choosing the node closest to the root of its tree, which produces an even distribution of sampling (Figure 1). However, we use the ability to control search to create algorithms which fit our needs.

4.2 Search Strategy

Our goal is to compute a set of points on the intersection curves, link them into a polygonal approximation and triangulate the region bounded by the polygon (interior or exterior as appropriate). Doing this requires that we be able to find a set of isolated points on the intersection curves (dots) with a controllable sampling rate. We do this with a two-part search strategy. The first part is a breadth-first search which ensures that each region of parameter space that could possibly contain an intersection is subdivided to a minimum degree. After the breadth-first subdivision, we are left with a set of "live" regions of parameter space which could still contain intersections based on the interval arithmetic tests. Many of these live regions turn out to be false positives – regions which in fact do not contain any intersections despite overlapping bounding boxes. In the second stage of the algorithm, we resolve the false positives using depth-first subdivision. We either find a dot to witness the intersection, or prove that no intersection exists. Figure 1 illustrates the results of the second search phase. The algorithm has proven that many of the live regions of parameter space in figure 2 really do not contain any intersections. In each of the remaining regions, the algorithm provides a dot and its mate in the other parameter space.

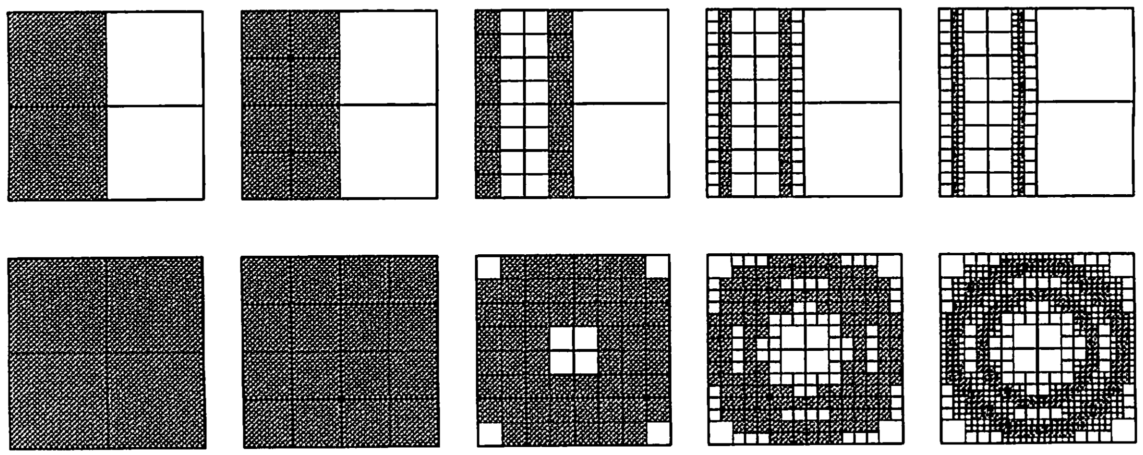


Figure 1: Stages in the breadth-first refinement of the intersection of a torus and a plane. On top is a display of the parameter space of the torus, on the bottom is the plane. All nodes are enclosed by squares. Leaf nodes with non-empty intersect lists are filled with grey.

Beginning with breadth-first subdivision and continuing with depth-first subdivision allows us to control dot spacing and dot accuracy separately. The level of the breadth-first subdivision controls the spacing, and the level of the depth-first subdivision controls the accuracy. This separate control is extremely valuable in practical situations, and is lacking in many algorithms.

5 Stringing and Triangulating

The interval refinement algorithm presented in the preceding sections provides a bounding region on the curves and points on the curves. In this section, we consider the problem of connecting these points together to build a polygonal representation of the curve and to triangulate only a part of parameter space bounded by these curves, to provide a "trimming" operation of cutting one surface against another.

The output of the interval refinement algorithm could be used to drive a marching method intersection. The points provide starting locations, and the bounding regions could help control the search. However, we are interested in directly applying the found points since we assume the user has specified tolerances which will provide a sufficient number of points to be found.

The first thing to notice about the dot connection problem is that the solution is not uniquely determined by the positions of the dots and their surrounding regions. Figure 3 illustrates the kinds of ambiguity which can arise. In order to string the dots into a chain, we must make further assumptions about the underlying intersection curve.

In stringing the intersection points together, we as-

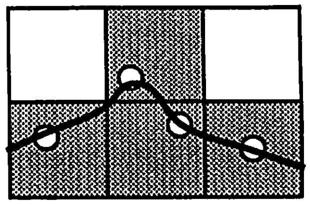


Figure 4: A "bump" is common case where our stringing assumptions fail. It is easy to create a hueristic which handles this special case.

sume that the intersection curve has low curvature relative to the grid size, and that different curves are always seperated by a grid cell at all points along their length. If we are interested in curves which do not cross, this restriction is acceptable if we pick a sufficiently small grid size. Under these assumption, an intersection curve will almost always pass through a cell exactly once, entering and exiting through different sides. Each grid cell which contains part of the intersection will be adjacent to exactly two others, unless it is at an edge. It is straightforward to connect the dots in this case.

Even if the intersection curve is well-behaved, quantization errors can cause the two neighbor assumption to be violated at any grid resolution. An example of such an error is the "bump" shown in Figure 4. Fortunately, this type of situation is not too difficult to deal with. If we remove the top-most dot in figure 4, the two-neighbor condition is restored and it is easy to connect the dots. Our stringer identifies such situations and removes dots to resolve the ambiguity.

Although we are unable to provide strong guarantees

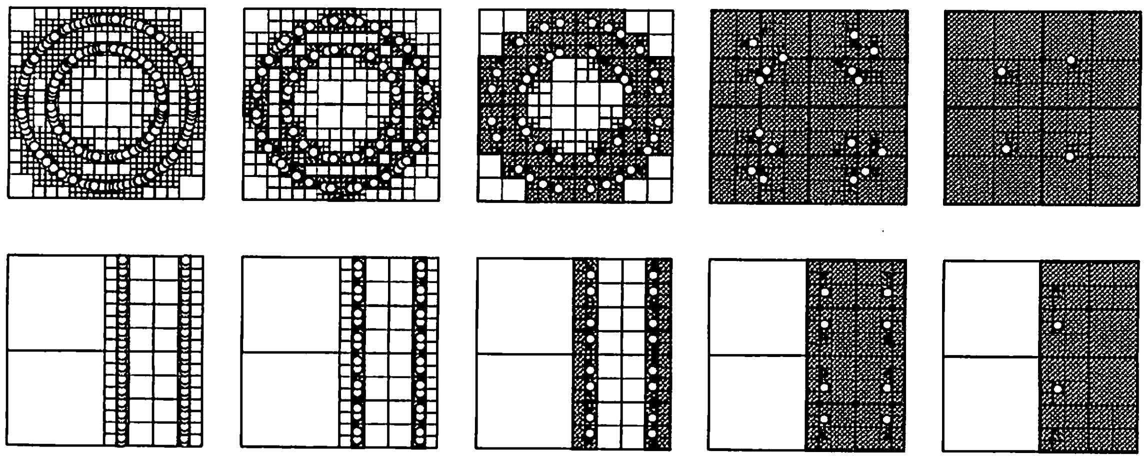


Figure 2: Dot finding is applied to the example of Figure 1. The white circles represent dots. For each gray square in Figure 1, a search for a dot was executed. If no dot was found, the square is rejected, and is not shaded.

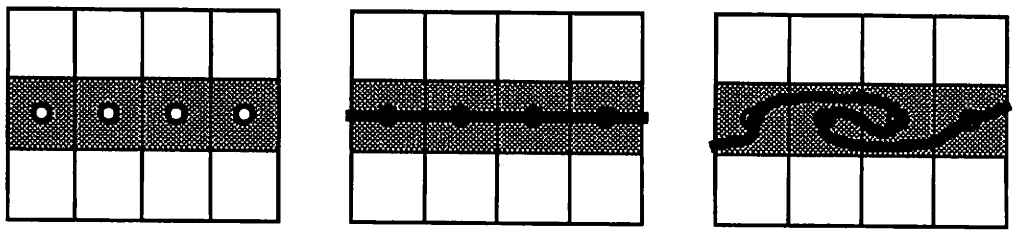


Figure 3: Although the stringing order for a set of dots may seem obvious, the curve may actually do something else. Without making further assumptions, points and bounding regions cannot uniquely determine a stringing order.

about the robustness of our stringer, it has performed well in our limited tests. Obviously, if the intersection is not a curve, but rather some degenerate case such as a surface region, stringing will not succeed in building a curve. Curves which intersect, or meet with tangency, violate the two neighbor criteria. By delaying decisions about ambiguous cases, other parts of the curves can be built correctly, typically providing enough information to make the stringing decisions, or at least satisfy the user. In cases that remain ambiguous, our prototype implementation uses further depth-first subdivision to verify the dot positions. We continue to explore other ways to resolve ambiguous situations with additional subdivision

Once the dots on each surface are connected, corresponding chains on each surface can be merged. This is important since we want to have one description of the curve which has the property that it includes a dot in each grid cell of both parameter spaces through which it passes.

5.1 Triangulating Trimmed Surfaces

One of our motivations for performing intersection calculations is to create trimmed surfaces which can be sewn together. In such cases it is crucial that the pieces can be assembled and mate together without "cracking" at the seams. In order for surfaces to fit together without cracks, their edges (as space curves) must be identical.

Our intersection method provides us with chains of dots in the parameter space of each surface which form piecewise linear approximations to the edges of the trimmed surface. If we triangulate two regions of different parameter spaces bounded by the same chains, the triangles will match without cracking, as shown in figure 5.

In our prototype implementation, we use a flood fill to place triangles in each grid square bounded by the intersection curves and then march around the intersection curve to fill in small triangles around the edge. This simple strategy has the disadvantage that it is not adaptive. To get sufficient detail around complicated intersections, we must also create large numbers of triangles in other areas of the surfaces. Techniques for doing adaptive subdivision could easily be added to our triangulation scheme, and in fact can employ information provided by the interval evaluations.

6 Status and Directions

We have implemented interval arithmetic as part of our mathematical toolkit in C++[8]. The algorithm described in this paper has been implemented and incorporated in an interactive scene composition program (used to create figures 5, 6, and 7) and in a mathematical modeling system [10].

Figures 5, 6, and 7 show results of our algorithm. In figure 5, the intersection curve is used to cut the sphere and cone to create a boundary representation of the boolean subtraction. The front row shows the results for three different levels of initial breadth-first search. Note that since dot size is controlled independently of spacing, even coarse tesselations with few points on the intersection curve still yield objects without cracks. The boolean union is rendered in the rear center. On each side, the separate objects are rendered with textures showing their parameterizations. The red curve on the texture is the trimming curve found by the algorithm. We have rotated the sphere so that its trimming curve is visible. The checkerboard pattern shows the cells that would be created by a uniform subdivision of parameter space to the level used in the breadth-first search of the front-center subtraction. Note that the actual breadth-first subdivision is not uniform. The blue squares are breadth-first subdivisions through which the trimming curve passes.

In figure 6, a sphere whose radius is modulated with a sinusoid is intersected with a torus. Figure 7 shows the intersection of a $sin(r)/(r+\epsilon)$ height field with a sphere. In both cases, the objects have been texture mapped in the same manner as figure 5. Figure 7 shows the union from top and bottom.

Although the prototype has been useful for creating a wide variety of interesting objects, there are several extensions to the basic algorithm which would be interesting to explore. For example, the search and triangulation techniques could be made to be non-uniform and adaptive to improve performance on surfaces with varying levels of detail. Stringing could be improved by using further subdivision to disambiguate difficult cases.

The surface intersection technique presented here can handle a very wide variety of surfaces because of its reliance on interval arithmetic. The algorithm progressively refines its model of the curve until user specified tolerances are met allowing separate control over the spacing and accuracy of the intersection points. Stringing the points together and triangulating the resulting parameter-space regions makes it possible to construct crack-free boundary representations of objects which are difficult to create with other methods.

Acknowledgements

This research was conducted at Apple Computer and at Carnegie Mellon. At Carnegie Mellon, this research is supported in part by a grant from Apple Computer and by an equipment grant from Silicon Graphics Incorporated. The first author is supported in part by a fellowship from the Schlumberger Foundation. Photorealistic RenderMan software, which was used to create images in this paper, was provided to Carnegie Mellon through Pixar's Renderman Education Program.

References

[1] Jarmo Alander. On interval arithmetic range approximation methods of polynomials and rational functions. *Computers and Graphics*, 9(4):365–372, 1985.

[2] G. Alefeld and J. Herzberger. *Introduction to Interval Computations*. Academic Press, 1983.

[3] C. Asteasu and A. Orbegozo. Parametric piecewise surfaces intersection. *Computers and Graphics*, 15(1):9–13, 1991.

[4] R. E. Barnhill, G. Farin, M. Jordan, and B. R. Piper. Surface / surface intersection. *Computer Aided Geometric Design*, 4:3–16, 1987.

[5] John J. Chen and Tulga M. Ozsoy. An intersection algorithm for C2 parametric surface. In Alison Smith, editor, *CAD86: Seventh International Conference on the Computer as a Design Tool*, pages 69–77. Butterworths, September 1986.

[6] Koun-Ping Cheng. Using plane vector fields to obtain all the intersections curves of two general surfaces. In *Theory and Practice of Geometric Modelling*, pages 187–204. Springer-Verlag, 1989.

[7] Daniel Filip, Robert Magedson, and Robert Markot. Surface algorithms using bounds on derivatives. *Computer Aided Geometric Design*, 3:295–311, 1986.

[8] Michael Gleicher and Andrew Witkin. Snap together mathematics. In Edwin Blake and Peter Weisskirchen, editors, *Advances in Object Oriented Graphics 1: Proceedings of the 1990 Eurographics Workshop on Object Oriented Graphics*. Springer Verlag, 1991. Also appears as CMU School of Computer Science Technical Report CMU-CS-90-164.

[9] Elizabeth G. Houghton, Robert F. Emnett, James D. Factor, and Chaman L. Sabharwal. Implementation of a divide-and-conquer method for intersection of parametric surfaces. *Computer Aided Geometric Design*, 2:173–183, 1985.

[10] Michael Kass. CONDOR: constraint-based data flow. *Computer Graphics*, 26, July 1992. to appear.

[11] Sheldon Katz and Thomas Sederberg. Genus of the intersection curve of two rational surface patches. *Computer Aided Geometric Design*, 5:253–258, 1988.

[12] Gabor Lukacs. The generalized inverse matrix and the surface-surface intersection problem. In *Theory and Practice of Geometric Modelling*, pages 167–185. Springer-Verlag, 1989.

[13] Don P. Mitchell. Robust ray intersection with interval arithmetic. In *Graphics Interface*, pages 68–72, 1990.

[14] Michael Mortenson. *Geometric Modelling*, chapter 7: Intersections, pages 319–344. John Wiley & Sons, 1985.

[15] S. P. Mudur and P. A. Koparkar. Interval methods for processing geometric objects. *IEEE Computer Graphics and Applications*, pages 7–17, February 1984.

[16] M. J. Pratt and A. D. Geisow. Surface/surface intersection problems. In J. A. Gregory, editor, *The Mathematics of Surfaces*, pages 117–142. Clarendon Press, 1986.

[17] Aristides G. Requicha. Representations for rigid solids: Theory, methods, and systems. *Computing Surveys*, 12(4):437–464, December 1980.

[18] T. W. Sederberg and R. J. Meyers. Loop detection in surface patch intersections. *Computer Aided Geometric design*, 5:161–171, 1988.

[19] Thomas Sederberg and Tomoyuki Nishita. Geometric Hermite approximation of surface patch intersection curves. *Computer Aided Geometric Design*, 8:97–114, 1991.

[20] Kevin G. Suffern. Interval methods in computer graphics. *Computers and Graphics*, 15(3):331–340, 1991.

[21] Daniel Toth. On ray tracing parametric surfaces. *Computer Graphics*, 19(3):171–179, July 1985.

[22] Brian Von Herzen and Alan Barr. Accurate triangulations of deformed, intersecting surfaces. *Computer Graphics*, 21(4):103–108, July 1987. Proceedings SigGraph '87.

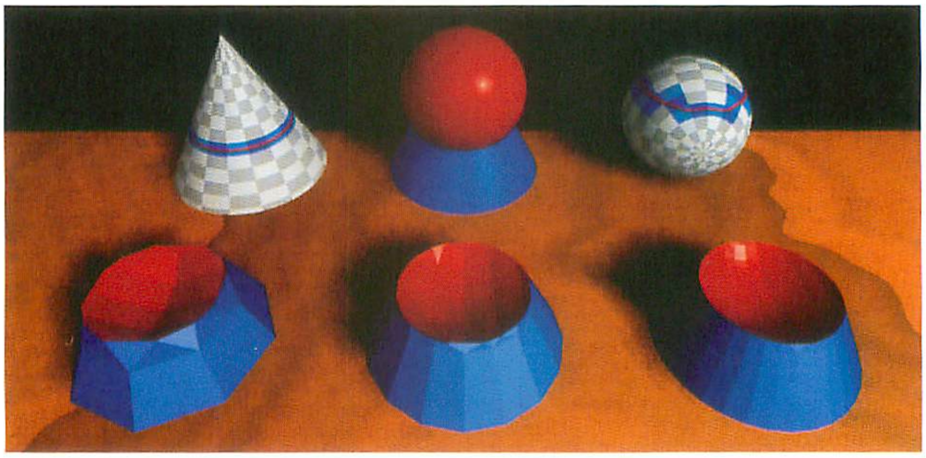


Figure 5: Trimming a cone against a sphere. Note that even coarse tesselations are crack-free.

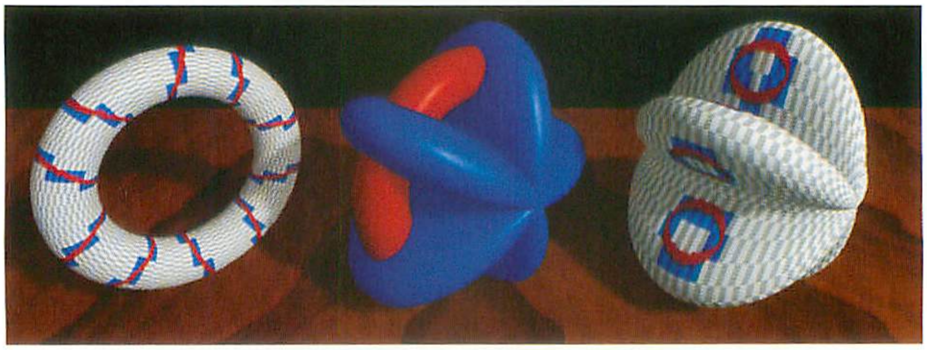


Figure 6: Intersection of a five-lobed surface and a torus.

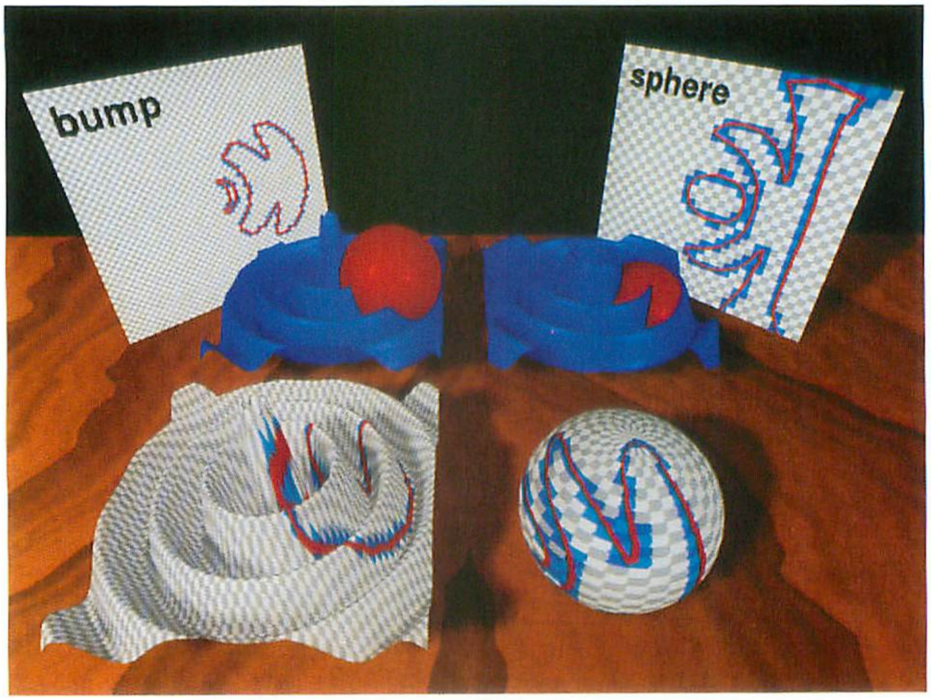


Figure 7: Intersecting a sphere with a $sin(r)/r$ bump. The middle right is a bottom view of the middle left.

Physically-Based Methods for Polygonization of Implicit Surfaces

Luiz Henrique de Figueiredo[†]
Jonas de Miranda Gomes[†]
Demetri Terzopoulos[‡]
Luiz Velho[†‡]

[†] IMPA – Instituto de Matemática Pura e Aplicada
Estrada Dona Castorina, 110, Rio de Janeiro, Brazil, 22460

[‡] University of Toronto – Department of Computer Science
Toronto, Ontario, M5S-1A4

Abstract

We present discrete physically-based methods for generating polygonal approximations of implicit surfaces. These methods not only generate a combinatorial manifold approximating the surface, but also produce a structure that is well suited to numerical simulations in physically-based modeling and animation systems.

Keywords: implicit models, physically-based models, polygonization, triangulation, domain decomposition.

1 Introduction

Consider a differentiable function $F : \mathbf{R}^n \to \mathbf{R}$, for which 0 is a regular value. This means that the gradient vector

$$\nabla F(p) = \left[\frac{\partial f}{\partial x_1}(p), \frac{\partial f}{\partial x_2}(p), \cdots, \frac{\partial f}{\partial x_n}(p)\right] \qquad (1)$$

is non-zero at all points p in the inverse image $M = F^{-1}(0)$. In this case, the set M is a differentiable manifold of dimension $n-1$ that we shall simply call an *implicit manifold* (Spivak, 1965).

Recently the use of implicit surfaces has attracted the attention of researchers in geometric modeling. Implicit surfaces are suitable for applying visualization techniques based on ray-tracing (see (Hanharan, 1983) (Barr, 1986)), but some difficulties arise when we try to sample or structure points on them in order to gain more information about their topology and geometry (Figueiredo, 1991). One of the important issues in this sampling and structuring problem is the computation of polygonal approximations to the surface. Polygonal approximations enable us to use the fast, special purpose processors of graphic workstations in order to display implicit surface models.

1.1 Polygonization of Implicit Surfaces

To capture the geometry of an implicit manifold, we must sample and structure points on it. In this paper, our objective is to structure the points in order to obtain a combinatorial manifold $\widetilde{M}$ that is close to M in some suitable topology. The manifold $\widetilde{M}$ is called a *polygonal approximation* of the surface M.

Polygonal approximations to implicit manifolds were first described in the classic paper (Allgower & Schmidt, 1985). The method proposed by Allgower and Schmidt consists of the following steps:

1. Compute a triangulation of the ambient space;
2. Replace the function F by its simplicial approximation $\widetilde{F}$ relative to this triangulation;
3. Refine the triangulation so that $\widetilde{F}$ is close to F. The combinatorial manifold is then obtained as the inverse image $\widetilde{F}^{-1}(0)$ of the simplicial approximation.

The Freudenthal triangulation is the simplest triangulation in $\mathbf{R}^n$: the space is subdivided into cubes and the triangulation is obtained by subdividing each n-cube into $n!$ simplices. Figure 1 shows a two dimensional example. For more details the reader should consult (Allgower & Georg, 1990).

Several variations of Allgower's method exist in the graphics literature (Wyvill *et al.*, 1986), (Loreson & Cline, 1987), (Bloomental, 1988), (Velho, 1989), (Hall & Warren, 1990). The correct computation of polygonal approximations to implicit manifolds depends on *a priori* estimates of the variation of the surface geometry (this is the refinement step (3) in the above algorithm). For this reason, some of the aforementioned works involve the computation of adaptive polygonizations in order to get better approximations.

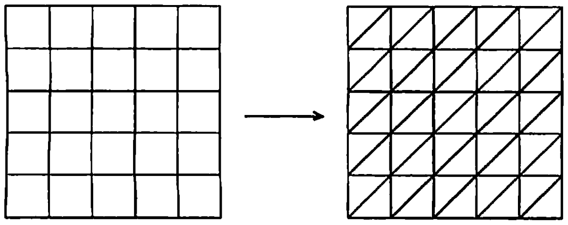


Figure 1: Freudenthal triangulation

1.2 Physically-based Modeling

Modeling is the most labor intensive part in the process of computer graphics. Modeling the motion of objects is often very difficult when the main goal is to generate realistic motion. The best solution to this problem is to model the physical habitat of the object: the motion will be a consequence of the interaction between the object and its environment, according to the laws of physics. A discussion of this physically-based modeling approach is found in several papers in the graphics literature (Barr *et al.*, 1987), (Terzopoulos & Fleischer, 1988).

1.3 Physically-based Polygonization of Implicit Objects

In this paper we use physically-based methods to compute polygonal approximations. These methods yield naturally adapted polygonizations. They also make it possible to construct a model such that the resulting polygonization has a natural physical structure associated with it which can be exploited for physically-based simulations.

Physically based methods in the study of implicit surfaces is a very recent research topic. In (Velho & Gomes, 1991) a spring-mass model is used to construct an adapted shell that approximates the geometry of the manifold. In (Velho & Gomes, 1991a), it is shown how this spring-mass shell can be used to do dynamical simulations with implicit models. In (Figueiredo, 1991), physically-based particle systems are used to sample points on an implicit manifold; algorithms for structuring such samples provide a powerful technique for modeling with implicit surfaces.

The physically-based approach to constructing piecewise linear approximation of implicit manifolds is related to the variational methods used to generate adaptive numerical grids for the numerical solution of partial differential equations (Thompson *et al.*, 1985). However, there are two main differences:

- To our knowledge, the adaptive methods in the numerical grid generation literature are developed for structured grids. The problem of polygonization of implicit surfaces is a non-structured one.
- In numerical grid generation, the physics of the associated problems may drive the adaptation of the grid. In our case, the primary interest is in the geometry and topology of the underlying grid space.

Our methods can certainly be used to generate adaptive numerical grids for problems where the physical domain can be defined implicitly. In fact, polygonization methods for implicit surfaces seem to be a very attractive technique for generating non-structured numerical grids.

In this paper, we are interested in polygonizations that are regular or "quasi-regular" triangulations. A *quasi-regular* triangulation is a 2-dimensional simplicial complex which is constituted by elements that are almost equilateral and equiangular. This type of polygonization is desirable in a number of applications to modeling and numerical simulation.

1.4 Overview

Section 2 describes the two discrete physical systems that we use to construct the polygonal approximation. Section 3 describes the polygonization algorithm using physically-based particle systems. Section 4 describes the polygonization algorithm using a spring-mass physical model. Section 5 gives examples and makes some comparisons between the two approaches described. Section 6 closes with a brief description of our current work in this area.

2 Discrete Physical Systems

A *discrete physical model* abstracts matter as an ensemble of particles related to each other by forces. Several physical phenomena may be naturally modeled using discrete physical systems (Greenspan, 1973). In a discrete physical system the particles interact under the action of internal and external forces. The associated motion equations are easily written as a classical $F = ma$ equation of Newtonian dynamics. Simple numerical integration methods, such as Euler's method generally produce good results.

In this work, we use two discrete physical models: a particle system and a spring-mass system.

2.1 Dynamic Particle Systems

A *particle system* is a finite set of particles which have an initial position in space and whose behavior in time is governed by algorithmic rules. Particle systems were introduced in graphics by Reeves as an algorithmic technique for modeling fire explosions (Reeves, 1983). In a *physical particle system*, the particles have masses and the Newtonian mechanics dictates their dynamical behavior. The motion of a particle depends on its mass, position and velocity, and on the forces acting on it, either by other particles or by the ambient medium. A *physical particle system* is a discrete physical model as defined above.

Physical particle systems have been used to simulate natural phenomena such as waterfalls (Sims, 1990) and fireworks (Weil, 1987). These systems in general require a significant amount of computational effort because of the number of particles involved. In Section 3 we shall use a simple physical particle system to compute a polygonal approximation to an implicit manifold. More recently, (Szeliski & Tonnesen, 1991) have applied physical particle systems to surface modeling.

2.2 Spring-Mass Systems

A *spring-mass system* is a physical particle system structured by connecting pairs of particles with springs. The springs impose *internal forces* that depend on the distance between these particles and govern the global behavior of the system. The resulting structure can be represented as a graph, where each particle is a node, and two nodes are connected when there is a spring joining the corresponding particles. Conversely, each graph linearly embedded in the space is naturally associated to a spring-mass system — a duality that will be exploited in Section 4 for triangulations.

Spring-mass systems are suitable to create physically-based models of deformable objects for dynamical simulation (Haumann, 1987), (Terzopoulos *et al.*, 1989). In the recent paper (Terzopoulos & Vasilescu, 1991), a spring-mass system is applied to adaptive image sampling and surface reconstruction. This approach has several connections with our method.

3 Polygonization using Dynamic Particle Systems

In this section, we describe an algorithm for computing a polygonal approximation of an implicit manifold using a physically-based particle system (Figueiredo, 1991).

3.1 Sampling using Dynamic Particle Systems

To properly sample a geometric object we must compute enough points on it so that its geometry can be reconstructed from the samples within some tolerance. In the case of a manifold given implicitly by a differentiable function $F : \mathbf{R}^n \rightarrow \mathbf{R}$, such a computation requires finding several solutions of the equation $F(x) = 0$. Physically-based methods for the solution of nonlinear equations have been known for some time (Incerti et al., 1979), although it seems that the main interest then was in finding any one solution, and not the many solutions that sampling requires. Consequently, these methods have not been applied to geometric modeling.

The particle systems we use for sampling derive their dynamics from the potential function $|F|$. The particles will seek equilibrium positions on the manifold $F^{-1}(0)$ because these are positions of minimum potential energy. If the gradient of F is non-singular, then these are the only equilibrium positions.

This interpretation of the gradient of $|F|$ as a force field implies the following equation of motion for a unit mass particle:

$$\frac{d^2x}{dt^2} + \gamma\frac{dx}{dt} + \text{sign}(F)\nabla F = 0, \qquad (2)$$

where γ is a positive real number representing friction proportional to velocity. (Incerti et al., 1979) have proposed a similar differential equation for finding zeros of functions $\mathbf{R}^n \rightarrow \mathbf{R}^n$.

3.2 Structuring Samples

The samples obtained by simulating the physics of particle systems have no structure other than the equilibrium position of each particle. Moreover,the samples are not evenly distributed across the surface, but rather tend to concentrate around points of high curvature. While this could be exploited for investigations on the geometry of the surface, a polygonal approximation interpolating such samples will rarely be quasi-regular.

In order to obtain a quasi-regular approximation, the sample is subjected to a relaxation process similar to the one used by (Turk, 1991) and (Szeliski & Tonnesen, 1991): particles repel each other with an intensity that rapidly decreases as the distance between the particles increases. Moreover, the movement of each particle is constrained to stay close to the surface by projecting repulsion forces onto the tangent plane.

The result of this relaxation process is a more uniform sampling of the surface. The desired polygonal approximation is then obtained by computing the Delaunay triangulation associated with the points and choosing the triangles that approximate well the tangent planes at each of its vertices.

4 Polygonization using Spring-Mass Systems

In this section, we describe a method to construct a polygonal approximation to an implicit manifold using a spring-mass system.

4.1 Subordinated Triangulation

Initially we define a system of spring-mass elements associated with a Freudenthal triangulation of the space. Like the particle systems described in Section 3.1, this system is subjected to deformation forces derived from the gradient field of the implicit manifold. Its equilibrium position gives a triangulation of a region of the space that contains the manifold M and has the following properties:

- M is transversal to the triangulation;

- The simplices are quasi-regular;

- For each n-simplex σ that intersects M there exists a point $p \in M$ close to the barycenter of σ such that the tangent space of M at p is close to the support hyperplane of one of the faces of σ.

Figure 2 illustrates the properties above in two dimensions. A triangulation with these properties is said to be *subordinated to the surface M* (Velho & Gomes, 1991).

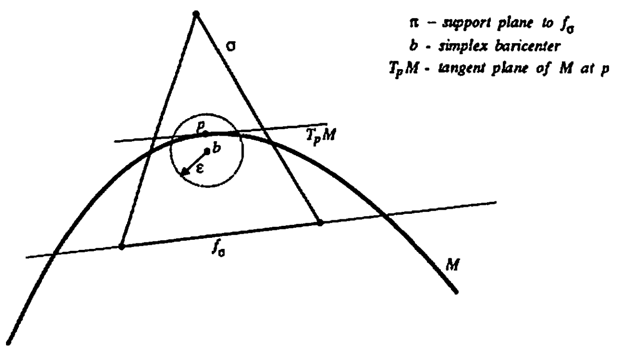

π - support plane to f_σ
b - simplex baricenter
T_pM - tangent plane of M at p
σ
p
b
ε
T_pM
f_σ
M

Figure 2: Subordinate triangulation

4.2 Mesh Generation

The spring-mass lattice generation process requires the following steps:

1. A Freudenthal triangulation is created within a volume bounding the implicit manifold;

2. Each simplex of the triangulation that intersects the implicit manifold is identified. Together, they form an intersecting simplicial complex;

3. A spring-mass system is created by associating mass nodes and springs to the vertices and edges of the intersecting complex.

The construction of the Freudenthal triangulation in step 1 is obtained as explained in Section 1.1. The identification of the relevant simplices in step 2 is done through a classification of the simplicial cells by testing the sign of the implicit function at the vertices of each simplex. Assuming that the uniform grid is sufficiently fine, if the signs are the same for all vertices, the simplex must be totally inside or totally outside of the manifold M. If the signs are different, then the simplex must intersect the surface M.

4.3 Mesh Deformation

After generating the mesh we use a physically-based approach in order to obtain the final triangulation that will be used for the polygonization of M. The dynamic simulation submit the spring-mass system to deformation forces with the purpose of conforming it to the shape of the implicit manifold. The process takes into account the internal forces produced by the springs as well as external deformation forces.

The external forces are based on information derived from the geometry of the implicit manifold. More specifically, two opposite attracting and repulsing force fields are generated using the gradient vector field of the implicit manifold. One field defined inside a small neighborhood of the object's boundary generates repelling forces that prevent points from being too close to the surface. The other force field, defined outside this neighborhood, generates attraction forces that pulls points towards the surface.

In order to facilitate the relaxation of the mesh structure into the desirable configuration, the initial rest length of the strings is made smaller than the initial grid spacing. This means that we start the process with a tensioned mesh that moves to a rest position under the action of internal and external forces.

4.4 Polygonization

The polygonization of the implicit manifold M is now obtained using the same technique of Allgower's algorithm described in Section 1.1: since the triangulation obtained is subordinated to M, the manifold intersects each 3-simplex σ in at most 4 distinct points, each one located on a different 1-dimensional face. Therefore, the linear approximation to M inside σ is formed by one or two triangles (2-simplices). The set of all these simplices constitute the combinatorial manifold that approximates M. We shall illustrate the method with some examples in section 5.1.

5 Results

In this section, we show the result of applying the two methods described in Sections 3 and 4 to compute polygonal approximations of implicit surfaces. We also make a comparative analysis of the polygonizations obtained and discuss the differences and similarities between the two methods.

5.1 Examples

Figures 3 and 4 illustrate the polygonization method using the particle systems presented in Section 3. Figure 3-a shows the trajectories of a particle system associated with a two-dimensional curve with 2 connected components described by the implicit equation $y^2 - x^3 + x = 0$. Figure 3-b shows the final equilibrium positions of these particles along the curve. Figure 4-a shows the sample points on the surface of the sphere $x^2 + y^2 + z^2 = 1$. Figure 4-b shows the polygonal approximation for the sphere.

Figures 5 to 7 illustrate the polygonization using the spring-mass system method presented in Section 4. Figure 5 demonstrates the mesh deformation process for the cylinder $x^2 + y^2 = 1$. Figure 5-a depicts the initial mesh created from a Freudenthal triangulation of the ambient space, Figure 5-b shows the final mesh in its equilibrium position. It is apparent that the mesh was constrained to lie in a tubular neighborhood of the implicit surface, conforming to the cylinder's shape. The polygonal approximation is obtained from this deformed mesh.

Figure 6 shows a detail of the polygonization associated with the spring-mass mesh before (a) and after (b) the deformation process. Note how the deformation of the mesh produces a very homogeneous polygon structure, transforming long, thin elements to nearly equilateral ones. This is because the triangulation resulting from the dynamical simulation is subordinate to the surface; as a consequence, the associated polygonization is quasi-regular.

Figure 7 shows the final polygonal approximation for the cylinder.

5.2 Comparisons

The main difference between the two methods presented in this paper is related to the order in which the operations of sampling and structuring of points on the implicit surface are performed.

The dynamical particle systems method in Section 3 first generates samples of the implicit object and subsequently structures these samples in order to create a polygonal approximation of the object.

The spring-mass systems method of Section 4 does the opposite. First the structure is created from a regular tessellation of space and second, this structure is used to sample the implicit object.

It is interesting to note that the physically-based approach is applied only to the sampling process. The structuring operation involves combinatorial methods.

The two methods produce equally good polygonal approximations of implicit surfaces. The combinatorial manifold generated by them is constituted by "almost fat" triangles.

The dynamical systems employed in both methods are very stable. The convergence to an equilibrium state is in general reasonably fast, requiring a small number of time steps (usually less than 100).

6 Conclusions

We have presented a new approach for the polygonization of implicit surfaces based on physically-based methods. The two methods described exploit different strategies to obtain polygonizations that are quasi-regular and faithfully approximate the original implicit objects.

The use of a physically-based approach for the polygonization of implicit objects provides great flexibility and control of the resulting structure.

Although this process is computationally more expensive than traditional methods, due to the numerical simulation of a dynamical system, it produces qualitatively better results.

We are presently incorporating these polygonization methods in a modeling and animation system for implicit objects.

Our current research also includes the development of adaptive physically-based polygonization methods and the application of these methods to numerical grid generation problems for domains defined by implicit surfaces.

In relation to the method of Section 3, we are investigating higher order approximations using intrinsic Voronoi diagrams. This would enable us to do continuous deformations using spline patches.

7 Acknowledgements

This research was partially supported by CNPq – Conselho Nacional de Desenvolvimento Científico e Tecnólogico, the Natural Sciences and Engineering Research Council of Canada and the Information Technology Research Center of Ontario.

8 References

Allgower, E. L. & Schmidt, P. H., (1985): An algorithm for piecewise-linear approximation of an implicitly defined manifold, *SIAM Journal of Numerical Analysis*, **22**, 322–346.

Barr, A., (1986): Ray Tracing Deformed Surfaces *Computer Graphics*, **20**, 4, 287-296, (Proceedings of SIGGRAPH '86).

Barr, A., Barzel, R., Haumann, D., Kass, M., Platt, J., Terzopoulos, D., Witkin, A., (1987): *Topics in Physically-based Modeling*, SIGGRAPH'87 course notes #17.

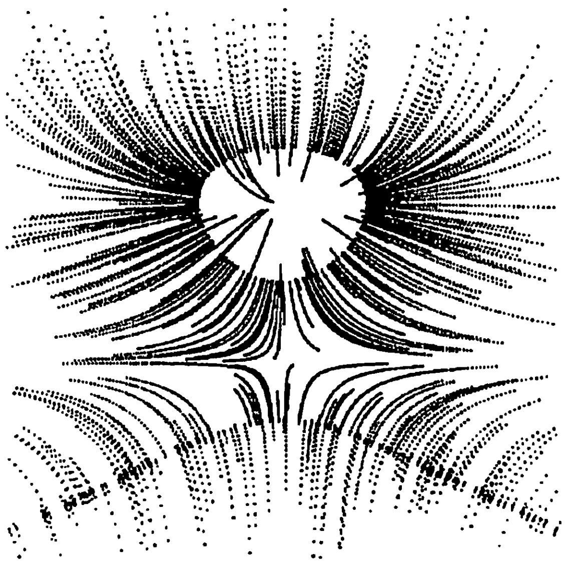
Bloomenthal, J., (1988): Polygonization of implicit surfaces, *Computer Aided Geometric Design*, **5**, 341–355.

Figueiredo, L. H., (1991): *Computational Morphology of Implicit Curves*, Ph.D. thesis in preparation, IMPA.

Greenspan, D., (1973): *Discrete Models*, Addison-Wesley, Reading, MA.

Hall, M. & Warren, J., (1990): Adaptive polygonization of implicitly defined surfaces, *IEEE Computer Graphics & Applications*, **10**, 33–42.

Hanharan, P., (1983): Ray Tracing Algebraic surfaces *Computer Graphics*, **13**, 1, 83-90, Proceedings of SIGGRAPH '83.


(a)

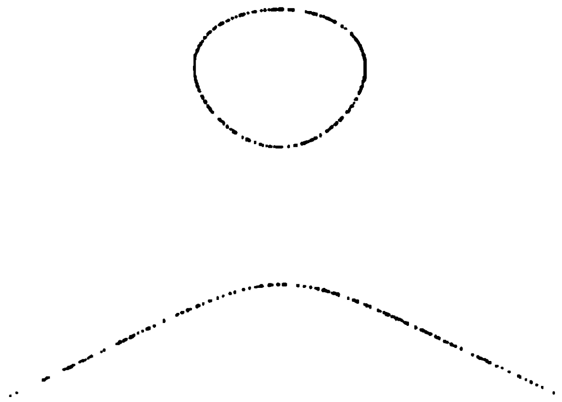
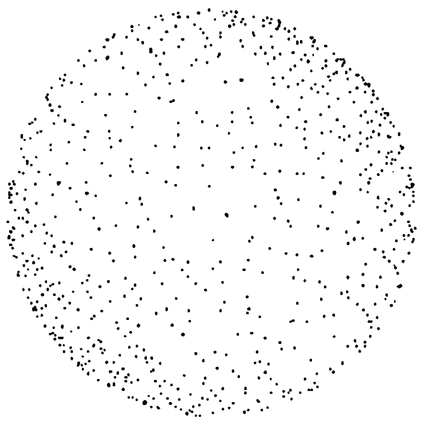

(b)

Figure 3: Trajectories (a) and final positions (b) of particles for 2D curve


(a)

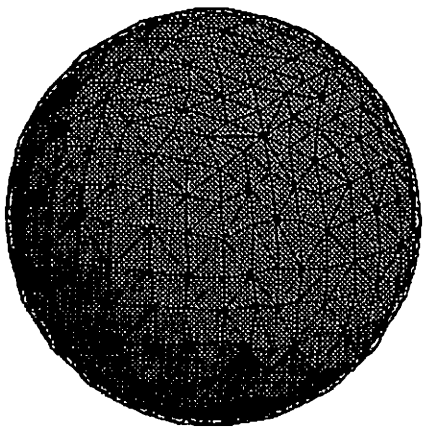
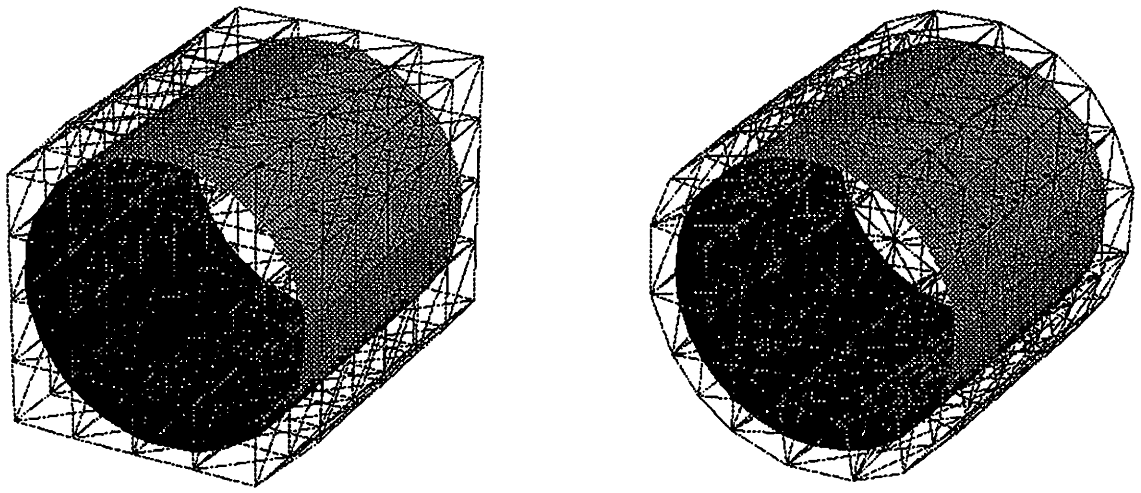
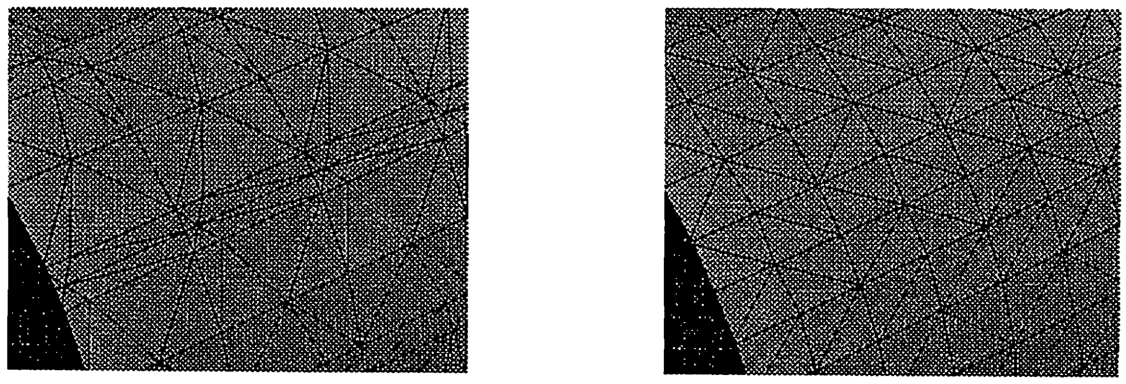

(b)

Figure 4: Sample points on the surface of a sphere (a) and polygonization of the sphere (b)



(a) (b)

Figure 5: 3D mesh before (a) and after (b) deformation



(a) (b)

Figure 6: Detail of the polygonization before (a) and after (b) deformation of the mesh

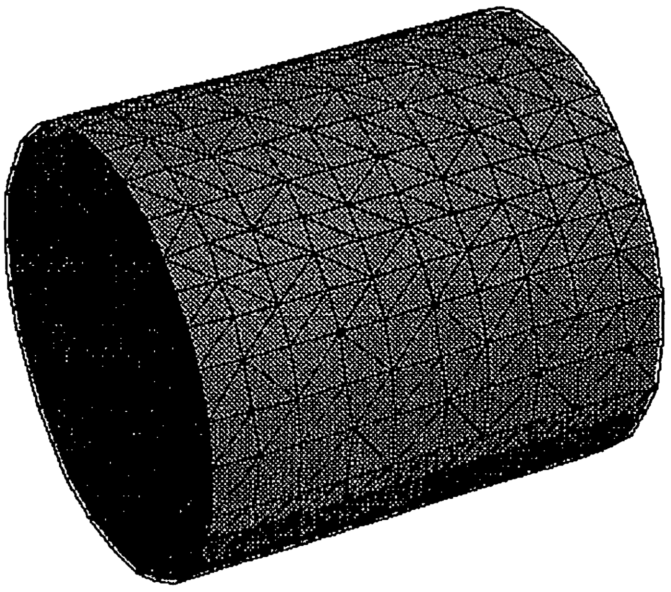


Figure 7: Final polygonization of the cylinder

Haumann, D., (1987): Modeling the physical behavior of flexible objects, in SIGGRAPH'87 course notes #17, 1-13.

Incerti, S., Parisi V., Zirilli, F. (1979): A new method for solving nonlinear simultaneous equations, *SIAM Journal on Numerical Analysis*, **16**, 779-789.

Loreson, W., & Cline, H., (1987): Marching cubes: a high resolution 3d surface construction algorithm, *Computer Graphics*, **21**, 163-169, (Proceedings of SIGGRAPH '87).

Reeves, W. T., (1983): Particle systems—a technique for modeling a class of fuzzy objects, *Computer Graphics*, **17**, 359-376, (Proceedings of SIGGRAPH '83).

Sims, K., (1990): Particle Animation and Rendering using Data Parallel Computation, *Computer Graphics*, **24**, 4, 405-414, (Proceedings of SIGGRAPH '90).

Spivak, M., (1965): *Calculus on Manifolds*, Benjamin.

Szeliski, R,. & Tonnesen, D., (1991): *Surface Modeling with Oriented Particle Systems*, CRL-91/14, Dec. 1991, Cambridge Research Lab-Digital Equipment Corporation, Cambridge, MA.

Thompson, J.F., Waisi, Z. U. Z., Mastin, C. W., (1985): *Numerical Grid Generation, Foundations and Applications*, North Holland, New York.

Terzopoulos, D., & Fleischer, K.; (1988): Deformable models, *The Visual Computer*,**4**, 306-331.

Terzopoulos, D., Platt, J., Fleischer, K., (1989): Heating and Melting Deformable Objects, *The Journal of Visualization and Computer Animation*, **2**, 2, 1991, 68-73. (also in Proceedings of Graphics Interface '89).

Terzopoulos, D. & Vasilescu, M., (1991): Sampling and Reconstruction with Adaptive Meshes, Proceedings of Computer Vision & Pattern Recognition Conference (CVPR-91), Lahaina, HI, 70-75.

Turk, G., (1991): Generating textures on arbitrary surfaces using reaction-diffusion, *Computer Graphics*, **25**, 289-298, (Proceedings of SIGGRAPH '91).

Velho, L. & Gomes, J. deM., (1991): Regular Triangulations of Implicit Manifolds using Dynamics, Proceedings of Compugraphics '91, Sesimbra, PT, 57-71.

Velho, L. & Gomes, J. deM., (1991a): A Dynamical Simulation Environment for Implicit Objects using Discrete Models, Proceedings of 2nd Eurographics Workshop on Animation and Simulation.

Weil, J., (1987): Boom Boom Boom, *(SIGGRAPH video review 1987)*, ACM SIGGRAPH, NY.

Wyvill, G., McPheeters, C., Wyvill, B. (1986): Data Structure for Soft Objects *The Visual Computer*, **2**, 4, 1986, 227-234.

Matrix Animation and Polar Decomposition

Ken Shoemake
Computer Graphics Laboratory
University of Pennsylvania
Philadelphia, PA 19104

Tom Duff
AT&T Bell Laboratories
Murray Hill, NJ 07974

Abstract

General 3×3 linear or 4×4 homogenous matrices can be formed by composing primitive matrices for translation, rotation, scale, shear, and perspective. Current 3-D computer graphics systems manipulate and interpolate parametric forms of these primitives to generate scenes and motion. For this and other reasons, decomposing a composite matrix in a meaningful way has been a long-standing challenge. This paper presents a theory and method for doing so, proposing that the central issue is rotation extraction, and that the best way to do that is Polar Decomposition. This method also is useful for renormalizing a rotation matrix containing excessive error.

Résumé

Des matrices correspondant à des transformations linéaires en 3 dimensions, ou bien à des transformations homogènes en 4 dimensions, peuvent être construites en composant des matrices qui décrivent des transformations élémentaires: déplacement, rotation, homothétie, glissement, et perspective. Les systèmes actuels de visualisation graphique a trois dimensions manipulent des formes paramétriques de ces transformations élémentaires, pour recréer des scènes et des mouvements. Il en découle l'intéret de trouver des décompositions pratiques de matrices composées. Nous présentons ici une technique pour trouver de telles decompositions. Le problème fondamental est l'extraction des rotations, et nous démontrons qu'une décomposition polaire est la méthode de choix. Cette méthode est aussi utile quant il faut renormaliser une matrice de rotation qui contient des erreurs excessives.

Keywords: homogeneous matrix, matrix animation, interpolation, rotation, matrix decomposition, Polar Decomposition, QR Decomposition, Singular Value Decomposition, Spectral Decomposition, greedy algorithm

Introduction

Matrix composition is well established as an important part of computer graphics practice and teaching [Foley 90]. It is used to simplify and speed the transformation of points, curves, and surfaces for modeling, rendering, and animation. Matrix decomposition—the focus of this paper—is less well known in computer graphics. It is useful for a variety of purposes, especially animation and interactive manipulation.

The usual transformations of an object can be described by 3×4 affine matrices; but the 12 entries of such a matrix are not very meaningful parameters. To understand, much less modify, matrices requires a good decomposition. Any decomposition must account for all 12 degrees of freedom (16 for 4×4 matrices) in the independent parameters of the primitives used. A decomposition that provides too few parameters will not be able to handle all inputs, while one that provides too many will not be stable and well-defined. The greatest problem, however, is ensuring that the decomposition is meaningful.

Most widely used 3-D animation systems, typified by Stern's *bbop* at NYIT [Stern 83], Gomez's *twixt* at Ohio State [Gomez 84] and Duff's *md* at Lucasfilm (later Pixar) allow the parameters of primitive transformations to be set interactively at key times, and compute transformations at intermediate times by spline interpolation in parameter space. Sometimes, however, only a composite matrix is available at each key frame, or more design flexibility is needed than that allowed by a hierarchy of primitive transformations. It is possible to interpolate the entries of a composite matrix directly, but the results are usually unsatisfactory. Decomposition allows the use of standard interpolation methods, and can give much better results. Matrix animation is discussed in more detail below.

Most authors have considered decomposition with less stringent criteria than ours. A common motivation is the need to synthesize an arbitrary matrix from a limited set of primitives, without regard for meaningfulness of the decomposition [Thomas 91]. Typically, these methods rely on a sequence of shears [Greene 86], and give factors that depend on the coordinate basis used. Shears are one of the less common operations in graphics, and a sequence of shears is a poor choice for animation. In contrast, the decomposition we propose has a simple, physical, coordinate independent interpretation, preserves rigid body motion as much as possible, and animates well.

Composition and Decomposition

Three types of matrix are commonly used for 3-D graphics: 3x3 linear, 3x4 affine, and 4x4 homogeneous; similar types with one less column and row are used for 2-D graphics. The homogeneous matrix is most general, as it is able to represent all the transformations required to place and view an object: translation, rotation, scale, shear, and perspective. Any number of transformations can be multiplied to form a composite matrix, so that a point can be transformed from model space coordinates to screen space coordinates in a single step.Generally, however, perspective is treated as a separate step in the viewing process—because lighting calculations must be done first—and not used for modeling or object placement. All the transformations except perspective can be accomodated by an affine matrix, which, in turn, can be considered just a 3x3 linear matrix with a translation column appended. (Following [Foley 90], we write points as column vectors, $[x\ \ y\ z\ \ 1]^T$, which are multiplied on the left by the matrix.)

$$\mathbf{P} = \begin{pmatrix} 1 & 0 & 0 & 0 \\ 0 & 1 & 0 & 0 \\ 0 & 0 & 1 & 0 \\ x & y & z & w+1 \end{pmatrix} \qquad \mathbf{T} = \begin{pmatrix} 1 & 0 & 0 & x \\ 0 & 1 & 0 & y \\ 0 & 0 & 1 & z \\ 0 & 0 & 0 & 1 \end{pmatrix}$$

$$\mathbf{R} = \begin{pmatrix} 1-2(y^2+z^2) & 2(xy-wz) & 2(xz+wy) & 0 \\ 2(xy+wz) & 1-2(x^2+z^2) & 2(yz-wx) & 0 \\ 2(xz-wy) & 2(yz+wx) & 1-2(x^2+y^2) & 0 \\ 0 & 0 & 0 & 1 \end{pmatrix}$$

$$\mathbf{K} = \begin{pmatrix} x & 0 & 0 & 0 \\ 0 & y & 0 & 0 \\ 0 & 0 & z & 0 \\ 0 & 0 & 0 & 1 \end{pmatrix} \qquad \mathbf{H} = \begin{pmatrix} 1 & h_{xy} & h_{xz} & 0 \\ h_{yx} & 1 & h_{yz} & 0 \\ h_{zx} & h_{zy} & 1 & 0 \\ 0 & 0 & 0 & 1 \end{pmatrix}$$

Figure 1. Primitive Transformation Matrices

Each primitive transformation has a more meaningful and concise representation than its matrix: a vector for most, a quaternion for rotations. It is not too difficult to recover the concise form if the matrix for the primitive is available [Goldman 91][Shoemake 91]. Once primitives have been multiplied into a composite matrix, however, recovery is generally impossible. Even so, a great deal can be done, as we shall see.

Primitive recovery is difficult for three reasons: absorption, order, and interaction. The first two problems are intractable; the third is the focus of this paper. Absorption is simple: a sequence of translations gives a result which is indistinguishable from a single translation, or from any number of different sequences; the same is true of other primitives. Order is also simple: the effect of a translation followed by a scale could as easily be achieved by composing primitives in the opposite order; likewise for other pairs. Interaction is more subtle: most transformations change all columns of the matrix, so scaling (for example) affects translation; all pairs of primitives interact. Notice any shear can be achieved by combining rotation and scale.

While absorption and order cannot be unscrambled, they can be standardized; for animation and other applications of interest, this usually suffices. Absorption can simply be ignored; that is, no attempt is made to tease apart a translation (except perhaps into x, y, and z components). Order is handled most easily by assuming a canonical order, such as Perspective of Translation of Rotation of Scale of object. Which canonical order is chosen is partly a matter of taste; this particular one makes translation trivial to extract, and places perspective in the order expected for a transformation to camera coordinates. If more information is made available in a particular situation, it may be possible to improve upon these standard assumptions; for example, it may be known that only x translation took place, or that scaling was done last. Such special case extraction is outside the scope of this paper.

Rigidity and Rotation

A perspective matrix of the form given above is easy to extract as a left factor of a composite homogeneous matrix, $\mathbf{C} = \mathbf{PA}$, with non-singular 3x3 corner; the details are left as an exercise for the reader. Notice that the usual perspective matrix includes translation and scale; we have chosen the minimal form necessary to reduce $\mathbf{C}$ to an affine matrix.† Likewise, a translation is easy to extract as the left factor of the remaining affine matrix, $\mathbf{A} = \mathbf{TM}$; simply strip off the last column. The matrix $\mathbf{M}$ then essentially will be the 3x3 matrix of a linear transformation. It would be simplest not to factor $\mathbf{M}$ at all, but to animate its entries directly. The results of this overly simple approach are visually disconcerting, but worth investigating.

Direct matrix interpolation treats each component of the matrix separately, and creates intermediate matrices as weighted sums of nearby key matrices. For example, linear interpolation between keys $\mathbf{M}_1$ and $\mathbf{M}_2$ uses $(1-t)\mathbf{M}_1+t\mathbf{M}_2$, while cubic spline interpolation uses affine combinations, $\alpha_1\mathbf{M}_1+\alpha_2\mathbf{M}_2+\alpha_3\mathbf{M}_3+\alpha_4\mathbf{M}_4$, with $\alpha_1+\alpha_2+\alpha_3+\alpha_4 = 1$. The results of this approach are immediately deduced from the linearity of matrix multiplication.

> **Proposition:** A point transformed by a weighted sum of matrices equals the weighted sum of the transformed points; that is,
>
> $$(\sum_i \alpha_i \mathbf{M}_i)\, p = \sum_i \alpha_i (\mathbf{M}_i\, p).$$

An example of this behavior can be seen in Figure 2, where the chin and hat back move steadily along a line from initial to final position, as do all the points. (We will use planar examples because they are easier to interpret on the page, but illustrate the same issues as spatial examples.) Notice that the interpolated matrix twice becomes singular as the

† But a permutation matrix may also be needed to provide pivoting for what is, in effect, a block LU decomposition.

image appears to flip over. At any moment of singularity the image will collapse onto a line (or worse, a point).

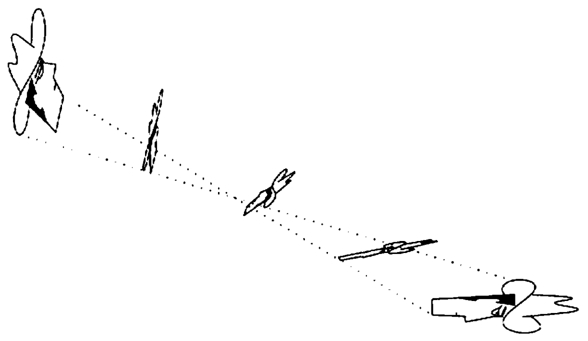


Figure 2. Direct Matrix Interpolation

Consider a square centered at the origin, and two key matrices: the identity and a 180° rotation. Since there are only two keys, only linear interpolation makes sense. Then, however, the theorem implies that each corner of the square will move linearly to its rotated position, which is diagonally opposite; the square will collapse through the origin. Although the distortions diminish with smaller angles of rotation, the square loses its shape. We expect rotations to transform the shape rigidly; direct matrix interpolation fails to do so. On the other hand, there is no problem with matrices for translation, scale, or shear.

Experiments with apparent motion (flash one image, flash another, see motion) suggest that the human visual system infers rigid motion as much as possible [Carlton 90] [Shepard 84]. Rotation is the only rigid transformation that is distorted by direct matrix interpolation. It therefore seems reasonable to conclude that the central problem for matrix animation is to extract a rotation in the best possible way, so that it can be interpolated *as* a rotation.

Decomposition Methods

Rotation matrices have simple defining properties: each column is a unit length vector which is perpendicular to the others, and the third column is the cross product of the first two. (Rows satisfy the same properties.) The first two properties are those of *orthogonality*, and can be summarized as $\mathbf{Q}^T\mathbf{Q} = \mathbf{I}$; the last makes the orthogonality *special*, and can be stated as $\det(\mathbf{Q}) = +1$. Orthogonality alone implies that the determinant must be either +1 or −1, with the latter indicating the presence of a reflection in the matrix. A 3x3 orthogonal matrix with negative determinant can be converted to a pure rotation by factoring out a $-\mathbf{I}$.

Numerical analysts have developed a number of algorithms for orthogonal matrices [Golub 89] [Press 88], in large part because orthogonality limits the accumulation of numerical error. Given a square—and presumably non-singular—matrix, three promising orthogonal decompositions are available: QR decomposition, Singular Value Decomposition (SVD), and Polar Decomposition. The QR factors of a matrix $\mathbf{M} = \mathbf{QR}$ are, respectively, orthogonal and lower triangular. The SVD gives three factors, $\mathbf{M} = \mathbf{UKV}^T$, with $\mathbf{U}$ and $\mathbf{V}$ orthogonal and $\mathbf{K}$ diagonal and positive. The less common Polar Decomposition, $\mathbf{M} = \mathbf{QS}$, yields an orthogonal factor and a symmetric positive definite factor. The latter two decompositions can factor singular matrices, with "positive" replaced by "non-negative" in the factors.

More than one algorithm is available to compute each decomposition. The oldest and best-known method for QR Decomposition is called Gram-Schmidt orthogonalization. Each row of the matrix is considered in turn, with each divided by its magnitude to give a unit vector, then projected onto the remaining rows to subtract out any parallel component in each of them. A better method is to accumulate Householder reflections, orthogonal transformations which can zero out the elements above the diagonal.

There is no simple SVD algorithm. The most common approach is first to use Householder reflections to make $\mathbf{M}$ bidiagonal, then to perform an iteration involving QR Decomposition until the off-diagonal entries converge to zero. While this is numerically reliable, it is complicated to code, and by no means cheap.

It is possible to compute a Polar Decomposition using the results of SVD, suggesting great cost; but a simpler method is available [Higham 86]. Compute the othogonal factor by averaging the matrix with its inverse transpose until convergence: Set $\mathbf{Q}_0 = \mathbf{M}$, then $\mathbf{Q}_{i+1} = \frac{1}{2}(\mathbf{Q}_i + \mathbf{Q}_i^{-T})$ until $\mathbf{Q}_{i+1} - \mathbf{Q}_i \approx \mathbf{0}$. This is essentially a Newton algorithm for the square root of $\mathbf{I}$, and converges quadratically when $\mathbf{Q}_i$ is nearly orthogonal. Finding the $\mathbf{Q}$ factor of a 2x2 matrix is easy. Suppose

$$\mathbf{M} = \begin{pmatrix} a & b \\ c & d \end{pmatrix};$$

then

$$\mathbf{Q} = \mathbf{M} + \operatorname{sign}(\det(\mathbf{M}))\begin{pmatrix} d & -c \\ -b & a \end{pmatrix},$$

scaled by a factor that makes the columns unit vectors.

Polar Decomposition Advantages

Care is needed in choosing among the possibilities, since the purposes of numerical linear algebra are different from those of computer graphics. The worst of the three choices seems to be SVD: it is the most expensive to compute, and the orthogonal matrices it produces are practically useless. A matrix which is already a pure rotation can be factored in an infinite variety of ways into the two orthogonal matrices of the decomposition, which is disastrous in the context of matrix animation. Small perturbations of the input matrix can cause different orthogonal factors to be chosen, even though the set of singular values is stable. Interpolating unreliable matrices will produce erratic results: consider the following two decompositions.

$$\begin{pmatrix}1&0&0\\0&1&0\\0&0&1\end{pmatrix}\begin{pmatrix}99.3&0&0\\0&99.4&0\\0&0&99.5\end{pmatrix}\begin{pmatrix}1&0&0\\0&1&0\\0&0&1\end{pmatrix}=$$
$$\begin{pmatrix}0&0&1\\1&0&0\\0&1&0\end{pmatrix}\begin{pmatrix}99.4&0&0\\0&99.3&0\\0&0&99.5\end{pmatrix}\begin{pmatrix}0&1&0\\0&0&1\\1&0&0\end{pmatrix}.$$

Although both are perfectly valid decompositions, interpolation of the primitives will give visible distortions —not at all what the user expects! Floating point variations in the least significant digit can cause an SVD algorithm to choose the first decomposition for one key, and the second for the next. Many SVD routines order the singular values by magnitude, which only exacerbates the problem. There seems to be no way to avoid having small input changes cause large output changes.

QR Decomposition is a much better choice, though it still presents problems. Unlike the factors of SVD, the QR factors can be determined uniquely, and are stable under small perturbations. Also, the algorithms for QR are simple and efficient. The drawback is that the orthogonal matrix extracted is not particularly meaningful: it is not independent of the coordinate basis used, and so has no "physical" significance. That is, if the matrix **M** is given in a rotated and uniformly scaled basis $\mathbf{M}' = \mathbf{BMB}^{-1}$, coordinate independent factors would have the form $\mathbf{Q}' = \mathbf{BQB}^{-1}$ and $\mathbf{R}' = \mathbf{BRB}^{-1}$; but the latter is no longer a lower triangular matrix, since that property is not preserved under similarity transforms. This is unfortunate for animation purposes, because it makes results much less predictable. Suppose, for example, **M** is constructed by rotating then scaling; although the **Q** factor might be expected to capture the rotation, it will not. Only when **M** is constructed by scaling then rotating will QR recover the original factors.

The Polar Decomposition factors are unique, coordinate independent, and simple and efficient to compute. Furthermore, the orthogonal factor **Q** is the closest possible orthogonal matrix to **M**, a property which is also coordinate independent. That is, **Q** satisfies the following conditions.

Find $\mathbf{Q}$ minimizing $\|\mathbf{Q}-\mathbf{M}\|_F^2$
subject to $\mathbf{Q}^T\mathbf{Q} - \mathbf{I} = \mathbf{0}$,

where the measure of closeness, the Frobenius matrix norm squared, is

$$\|\mathbf{Q}-\mathbf{M}\|_F^2 = \sum_{i,j}(q_{ij}-m_{ij})^2.$$

Since this important claim appears in [Higham 88] without proof, a proof is given in the Appendix. When **M** has positive determinant, **Q** will be a pure rotation, otherwise it will include a reflection. It might seem preferable to exclude reflections, but there is no well-defined nearest rotation. For example, every 2-D rotation is equally distant from every 2-D reflection. (Polar Decomposition is applicable to matrices of any size and shape.) A rotation has the form $\begin{pmatrix}c&-s\\s&c\end{pmatrix}$, with $c^2+s^2=1$, while a reflection is $\begin{pmatrix}a&b\\b&-a\end{pmatrix}$, with $a^2+b^2=1$. The sum of the squares of the differences is $(c-a)^2+(-s-b)^2+(s-b)^2+(c+a)^2 = 2(c^2+s^2)+2(a^2+b^2) = 4$. As noted earlier, however, a 3x3 **Q** matrix which includes a reflection (indicated by a negative determinant) can be factored as $\mathbf{Q} = \mathbf{R}(-\mathbf{I})$.

Closeness also makes Polar Decomposition good for matrix renormalization. Moderate amounts of numerical noise can be removed in a single iteration of the averaging algorithm. This improves and formally grounds [Raible 90].

The combination of uniqueness and closeness guarantees that small input perturbations will not produce large output variations. The **Q** factor of Polar Decomposition appears to be the best possible rotation. What, then, is the **S** factor? As the appendix shows, in some rotated coordinate system **S** is diagonal—in other words, a scale matrix. This form of scaling is preserved through coordinate changes, and has a good claim to being a new primitive, *stretch*. The **S** factor can move to the other side of the **Q** factor without changing form, though its value will change to $\mathbf{Q}^T\mathbf{SQ}$. Thus Polar Decomposition has a very physical interpretation.

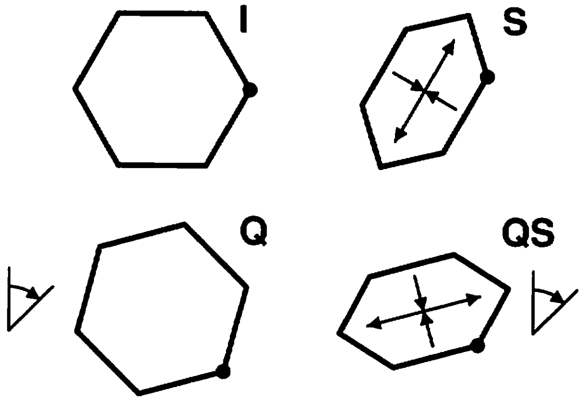


Figure 3. Physical View of Polar Decomposition

One drawback of Polar Decomposition is that there is no explicit representation of shear. As explained earlier, interaction is to blame; shear will be factored as rotation and stretch. In two dimensions, for example, a simple shear will factor as

$$\begin{aligned}\mathbf{H} &= \begin{pmatrix}1&h\\0&1\end{pmatrix}\\ &= \frac{1}{\sqrt{4+h^2}}\begin{pmatrix}2&h\\-h&2\end{pmatrix}\begin{pmatrix}2&h\\h&2+h^2\end{pmatrix}\frac{1}{\sqrt{4+h^2}}\\ &= \mathbf{QS}.\end{aligned}$$

As Figures 4 and 5 show, the appearance of a factored animation can be quite different from that of a direct animation for shear. Nevertheless, factorization gives a reasonable result.

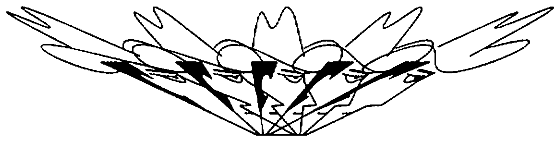


Figure 4. Direct Shear Interpolation

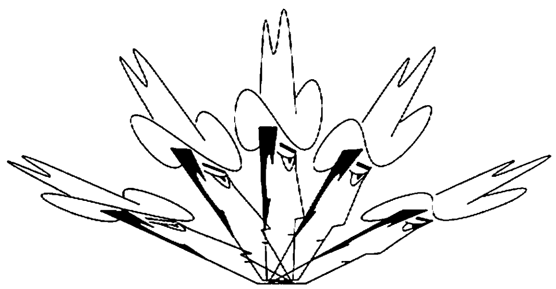


Figure 5. Decomposed Shear Interpolation

Direct Stretch Animation

Although S can be factored into diagonal form, $S = UKU^T$ (using a symmetric eigenvalue routine [Golub 89] [Carnahan 69]), as with SVD the factorization is not unique. This unavoidable indeterminacy combined with small numerical errors could cause different U's to be chosen at different keys, and the resulting interpolation would suffer greatly. Fortunately, however, S matrices can be interpolated directly, and will preserve their form and meaning. That is, $\alpha_1 S_1 + \alpha_2 S_2 + \dots$ yields a symmetric matrix, which for non-negative α_i will also be positive definite, so it is not necessary to choose a diagonalizing rotation U. If some U diagonalizes both S_1 and S_2 simultaneously, then $\alpha_1 S_1 + \alpha_2 S_2 = U(\alpha_1 K_1 + \alpha_2 K_2)U^T$, so direct interpolation simply interpolates the scale factors, as desired. Weights α_i for interpolation will usually include negative values (to ensure smooth motion), so the interpolated S matrices can become singular; but the same thing can happen with pure scale matrices. In both cases this does not seem to be a serious problem, and can be solved using spline tension.

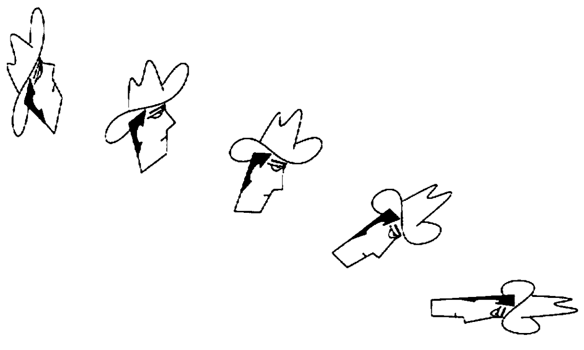


Figure 6. Polar Decomposed Matrix Interpolation

Factored Stretch Animation

Diagonalization of S as UKU^T is still a useful alternative if it can be stabilized across keys. (Even without stabilization, an interactive user interface will certainly deal with stretch in factored form.) So in this section—which can be skipped on first reading—we consider the following problem: Given two stretch matrices, S_1 and S_2, interpolated in that order, how can their diagonalizing rotations, U_1 and U_2, be chosen to be as similar as possible? More precisely, if the rotation taking U_1 into U_2 is designated by $U_{12} = U_1^T U_2$, the problem is to minimize the absolute angle of rotation performed by U_{12}. Furthermore, so that the results can easily be generalized to a series of matrices S_i, let U_1 be fixed. (Then fix U_2 while minimizing U_{23}, and so on. Begin with $U_0 = I$.)

There are three cases, depending on how many identical values occur on the diagonal K_2. When all three values are the same, it is possible to set $U_2 = U_1$. Uniform scaling is common in computer graphics practice, and is easily detected by inspection of S_2, which will already be diagonal with identical values. When all three values are different, we have 24 choices for U_2. These are obtained by all axis permutations (6), times all axis sign combinations (8), achievable by a rotation (divide by 2). When exactly two values are the same, we have an extension of the all different case: free rotation is allowed around one of the axes. The last two cases are discussed more fully below.

An easy way to measure the rotation U_{12} is to convert it into a unit quaternion. ([Shoemake 85] introduces unit quaternions as a representation of 3-D rotation and discusses how to interpolate them.) Its real component is $\cos(\theta/2)$, where θ is the total rotation angle. Picking U_2 to maximize the quaternion's real component minimizes the angle.

There is a quick way to do this maximization. Let q be the quaternion corresponding to U_{12}. The 24 variations correspond to qp, where p is one of 48 quaternions (including both p and $-p$) that map the coordinate axes into themselves: $p = [x\ y\ z\ w]$ can be one of $[0\ 0\ 0\ \pm1]$, $[0\ 0\ \pm1\ \pm1]/\sqrt{2}$, $[\pm1\ \pm1\ \pm1\ \pm1]/2$, or a permutation of these. The real part of qp is $w_q w_p - x_q x_p - y_q y_p - z_q z_p$, which is simple to maximize because of the simple form of each p. We take the absolute values of the components of q, sort them, and choose the maximum of either the largest, or half the sum of all four, or $1/\sqrt{2}$ times the sum of the two largest. Then we can work backwards from our choices to deduce the corresponding p.

If exactly two of $\mathbf{K}_2$'s values are the same, we have a continuous optimization. As before, we are free to permute the axes, but we have the additional freedom to rotate by any angle in the plane of equal scaling. We can arrange for the equal values to be the first two, so that a change of coordinates rotating around the z axis leaves $\mathbf{K}_2$ unchanged. So our problem is to pick p and r to maximize the real component of qpr, where p is one of the 48 quaternions above and $r = [0\ 0\ s\ c]$, with $c^2+s^2=1$, is a quaternion that rotates about the z axis.

The product of a quaternion $[x\ y\ z\ w]$ with $r = [0\ 0\ s\ c]$ is $[xc+ys\ \ yc-xs\ \ zc+ws\ \ wc-zs]$. Choosing $c = w/\sqrt{w^2+z^2}$ and $s = -z/\sqrt{w^2+z^2}$ maximizes the real component to $\sqrt{w^2+z^2}$. Consequently, the best p is one that maximizes w^2+z^2. Only six values of p give essentially different results. These are $[0\ 0\ 0\ 1]$, $[1\ 1\ 1\ 1]/2$, $[1\ 1\ 1\ -1]/2$ and each of these times $[1\ 0\ 0\ 0]$. Summing the squares of the w and z components from the product of q with each of these and subtracting $^1/_2$ gives $\pm(w^2+z^2-{}^1/_2)$, $\pm(xz-wy)$ and $\pm(wx+yz)$. Choose the p corresponding to the largest positive value, and if the negative sign was used, post-multiply p by $[1\ 0\ 0\ 0]$.

This method for stabilizing the S decomposition is a greedy algorithm. It extends partial solutions at each stage by finding an optimal continuation, with no backtracking. There is no guarantee that this produces a global optimum—a locally inferior choice could possibly be warranted because it allows better choices further on that more than compensate. However, we can prove the following:

> **Proposition**: Given a sequence $\mathbf{S}_i$ of symmetric, positive definite matrices, none of which has a diagonalization with exactly two equal values, the greedy algorithm given above picks a sequence of rotation matrices $\mathbf{U}_i$ that minimizes the sum of the rotation angles between adjacent rotations.

The proof depends on two observations. First, the $\mathbf{S}_i$ with three equal values do not affect the sum; and second, the axis-permuting rotations p form a group. With this in mind, let $\langle p_i \rangle$ be the greedy sequence of permutations, and $\langle P_i \rangle$ the optimal sequence. Suppose now that some $p_k \neq P_k$. Then the discrepency $\delta = P_k^{-1}\, p_k$ is in the group, and can post-multiply every P_i, $i \geq k$ without increasing the angle sum. For P_k is replaced by p_k, which by definition of the greedy sequence gives the smallest angle possible at that step; and none of the other angles change, since $\delta^{-1} q_i^{-1} q_{i+1} \delta$ has the same angle as the original $q_i^{-1} q_{i+1}$. So $\langle p_i \rangle$ is also optimal.

With double values, however, some greedy sequences are not optimal. In mitigation, we point out that floating-point arithmetic stands between us and any reliable determination of equality of values, and that the additional freedom offered by equal values only causes the greedy algorithm to find solutions with smaller total rotation. Furthermore, the global optimization problem in the general case is a mixed-integer programming problem of the sort that is often NP-complete. (But we make no claims as to the status of this particular problem.)

Lest this extended discussion leave the wrong impression, we point out that diagonalization has not been necessary in our experience. The animations achieved by direct **S** interpolation look as good as those using the more elaborate procedure. (Also, the code required is much shorter than the discussion.) Since the developer of an animation system may choose not to introduce our new stretch primitive, however, we have offered a reasonable alternative.

Conclusions

With the assistance of Polar Decomposition, a non-singular 4×4 homogeneous matrix **M** can be factored into meaningful primitive components, as

$$\mathbf{M} = \mathbf{PTRNS},$$

where **P** is a simple perspective matrix, **T** is a translation matrix, **R** is a rotation matrix, **N** is ±**I**, and **S** is a symmetric positive definite stretch matrix. The stretch matrix can optionally be factored, though not uniquely, as $\mathbf{UKU}^\mathrm{T}$, where **U** is a rotation matrix and **K** is diagonal and positive. For a 4x3 affine matrix the perspective factor can be dropped; and for a 3x3 linear matrix, so can the translation. Also, **N** can be multiplied into **S** if desired.

Polar Decomposition produces factors **QS** which are unique, coordinate independent, and both simple and efficient to compute. The factors have a physical, visual interpretation not found with other decomposition methods. The **PTRNS** decomposition is useful for a variety of purposes, including matrix animation and interactive interfaces. It has the minor disadvantage that it does not directly represent shear.

Acknowledgements

Craig Reynolds first raised this challenge with Ken. John Gilbert, Eric Grosse, and Eugene Salamin shared their wisdom regarding numerical techniques and linear algebra. We thank Ed Shonberg for providing our French translation.

Appendix

Theorem: The Polar Decomposition factor **Q** is the closest possible orthogonal matrix to **M**, with closeness measured using the Frobenius matrix norm. That is, **Q** satisfies the following conditions.

$$\text{Find } \mathbf{Q} \text{ minimizing } \| \mathbf{Q}-\mathbf{M} \|_\mathrm{F}^2$$
$$\text{subject to } \mathbf{Q}^\mathrm{T}\mathbf{Q} - \mathbf{I} = \mathbf{0},$$

where

$$\| \mathbf{Q}-\mathbf{M} \|_\mathrm{F}^2 = \sum_{i,j} (q_{ij}-m_{ij})^2 .$$

Proof: Though expressed in matrix terms, the proof simply requires finding the minimum of a quadratic function, which we learned to do in calculus by finding where the derivative is zero. We can express $\|M\|_F^2$ as the diagonal sum—the trace—of M^TM, and incorporate the orthogonality constraint as a linear term using a symmetric Lagrange multiplier matrix Y. So, as the reader can verify, we can differentiate

$$\text{trace}\,[\,(Q-M)^T(Q-M) + (Q^TQ - I)Y\,]$$

with respect to Q and equate to zero to obtain

$$2(Q-M) + 2QY = 0$$

which simplifies to

$$Q(I+Y) = M.$$

Thus M will be factored as our desired Q times a symmetric $S = I+Y$.

$$M = QS.$$

This factorization is the Polar Decomposition of M. To use it we need to solve for S in terms of M. Since $Q^TQ = I$, we must have

$$(MS^{-1})^T(MS^{-1}) = I.$$

A symmetric S has a symmetric inverse, so this simplifies to

$$S^{-1}M^TMS^{-1} = I,$$

and finally to

$$S^2 = M^TM.$$

Now, M^TM is guaranteed to be symmetric and positive definite (or semi-definite if M is singular), and so there is a similarity transform that makes M^TM diagonal, with positive (or zero) real entries. This gives the Spectral Decomposition of S^2.

$$S^2 = UKU^T; \quad U^TU = I, \quad K = \begin{pmatrix} \kappa_1 & 0 & 0 \\ 0 & \kappa_2 & 0 \\ 0 & 0 & \kappa_3 \end{pmatrix}; \quad \kappa_i \geq 0.$$

Taking either the positive or negative square root of each diagonal element of K, we obtain eight candidates for S,

$$U\begin{pmatrix} \pm\sqrt{\kappa_1} & 0 & 0 \\ 0 & \pm\sqrt{\kappa_2} & 0 \\ 0 & 0 & \pm\sqrt{\kappa_3} \end{pmatrix}U^T.$$

However, for Q to be a minimal solution, the second derivative, $2(I+Y) = 2S$, of our function must be positive definite, which means only the positive square roots are allowed, and so S is uniquely determined. For any M which is non-singular, Q is also uniquely determined.

■

References

[Carlton 90] Carlton, Eloise H. and Shepard, Roger N. "Psychologically Simple Motions as Geodesic Paths," *Journal of Mathematical Psychology*, **34**(2), June 1990, 127–228

[Carnahan 69] Carnahan, Brice, Luther, H. A. and Wilkes, James O., *Applied Numerical Methods*, Wiley, 1969

[Foley 90] Foley, James D., van Dam, Andries, Feiner, Steven K., and Hughes, John F. *Computer Graphics: Principles and Practice, 2nd ed.*, Addison-Wesley, 1990

[Goldman 91] Goldman, Ronald N. "Recovering the Data from the Transformation Matrix," Gem VII.2, *Graphics Gems II*, Academic Press, 1991, 324–331

[Golub 89] Golub, Gene H., and Van Loan, Charles F. *Matrix Computations, 2nd ed.*, Johns Hopkins University Press, 1989

[Gomez 84] Gomez, Julian, "Twixt: a 3-d Animation System," *Proceedings of Eurographics '84*, Elsevier Science Publishers, 1984

[Greene 86] Greene, Ned, "Extracting Transformation Parameters from Transformation Matrices", NYIT, Personal communication

[Higham 86] Higham, Nicholas, "Computing the Polar Decomposition—With Applications", *SIAM J. Sci. and Stat. Comp.* 7(4), October 1986, 1160-1174

[Higham 88] Higham, Nicholas, and Schreiber, Robert S. "Fast Polar Decomposition of An Arbitrary Matrix," Technical Report 88-942, October 1988, Department of Computer Science, Cornell University

[Press 88] Press, William H., Flannery, Brian P., Teukolsky, Saul A., and Vetterling, William T., *Numerical Recipes in C*, Cambridge University Press, 1988

[Raible 90] Raible, Eric. "Matrix Orthogonalization," *Graphics Gems*, Academic Press, 1990, p. 464.

[Shepard 84] Shepard, Roger N. "Ecological Constraints on Internal Representation: Resonant Kinematics of Perceiving, Imagining, Thinking, and Dreaming," *Psychological Review*, **91**(4), October 1984, 417–447

[Shoemake 85] Shoemake, Ken. "Animating Rotation with Quaternion Curves," *Proceedings of SIGGRAPH '85 (San Francisco, California, July 22–26, 1985)*, In *Computer Graphics* **19**(3), July 1985, 245–254

[Shoemake 91] Shoemake, Ken. "Quaternions and 4x4 Matrices," Gem VII.6, *Graphics Gems II*, Academic Press, 1991, 351–354

[Stern 83] Stern, G., "Bbop—A System for 3d Keyframe Figure Animation," SIGGRAPH '83 Course Notes, Introduction to Computer Animation, 1983

[Strang 86] Strang, Gilbert. *Introduction to Applied Mathematics*. Wellesley-Cambridge Press, 1986

[Thomas 91] Thomas, Spencer W. "Decomposing a Matrix into Simple Transformations," Gem VII.1, *Graphics Gems II*, Academic Press, 1991, 320–323

Generating Natural-looking Motion for Computer Animation

Jessica K. Hodgins
Paula K. Sweeney
David G. Lawrence[†]
IBM T. J. Watson Research Center
Yorktown Heights, NY 10598

Abstract

The automatic generation of motion for animation remains an unsolved problem in computer graphics. One approach to the problem is to combine physically accurate models with control systems. The user specifies high-level goals and the control system computes the forces and torques that the simulated muscles or motors should exert to cause the model to perform the desired task. In this paper we describe control systems for rigid-body models of humans performing four tasks: pumping a swing, riding a seesaw, juggling, and pedaling a unicycle. We designed the control systems with the goal of producing natural-looking motion, and we discuss the techniques that we used to achieve this goal.

Keywords: Animation, Simulation, Control Theory.

Introduction

We would like to be able to automatically generate natural-looking motion for computer animations based on high-level input from the user. We have explored one solution to this problem: combining control systems with physically accurate models. The user specifies a high-level goal ("ride the unicycle from here to there") and the control system computes the forces and torques that will cause the simulated model to perform the desired task. The combination of a carefully designed control system and a realistic physical model can produce natural-looking motion that has much in common with the motion of the animal or human on which it was modeled.

Physical simulation has been used successfully for generating realistic motion of *passive systems* (Barzel and Barr 1988; Hahn, J. 1988; Terzopoulos and Fleischer 1988; Terzopoulos and Witkin 1988; Baraff 1989, 1991; Pentland and Williams 1989; Kass and Miller 1990; Norton et al 1991; Wejchert and Haumann 1991). Passive systems are those that are acted upon by the environment but have no internal source of energy. Bouncing balls, leaves blowing in the wind, and raindrops falling into puddles are all passive systems. To the extent that the computer program accurately models the physical system and the environment, the resulting motion will be natural. In contrast to passive systems, the systems described in this paper, *active systems*, contain simulated muscles or motors that provide an internal source of energy and allow the systems to act on the environment. Humans, animals, robots, and vehicles are all active systems. Simulation of active systems requires not only a physically realistic model of the system being animated but also a control system or computer algorithm that activates the muscles or motors in such a way that the system performs the desired task.

Because the internal workings of biological control systems are much less well understood than the physical systems themselves, the simulation of active systems is, in general, more difficult than that of passive systems. Active systems have been animated by using springs and dampers as the control system (Miller 1988), by programming a control system for a simplified model of the system being animated (Bruderlin and Calvert 1989), and by simulating the actions of the oscillators found in simple biological control systems (McKenna and Zeltzer 1990).

The design of control systems has not yet been automated for systems of the complexity of those that we would like to animate. Others have begun to address the question of automatic generation of motion in the context of optimization problems and optimal control theory (Witkin and Kass 1988; van de Panne, Fiume, and Vranesic 1990). The potential generality of these approaches makes them among the most interesting new methods for animation of dynamic systems. The potential liability is the growth of the search spaces when applied to more complex systems.

Our approach to hand designing these control systems builds on previous work in the control of legged robots (Raibert and Hodgins 1991; Hodgins 1991; Hodgins and Raibert 1990; Raibert 1986). This work provides us with a number of techniques that aid in the design of control systems for dynamic tasks: state machines for structuring the control laws, low-level control through springs and dampers, and symmetry of the motions as a principle for the design of the higher level control algorithms.

In this paper we describe control systems for rigid-body models of humans performing four tasks: pumping a swing, riding a seesaw, juggling, and pedaling a unicycle. In each case, the simulated models are composed of rigid links connected by rotary or sliding joints. The models are derived from measurements of living humans and cadavers (Meredith 1969a,b; Dempster and Gaughran 1965). Each model has enough degrees of freedom to perform the stipulated task but only a small fraction

[†]Present address: Department of Aeronautics and Astronautics, Stanford University, Stanford, CA 94305.

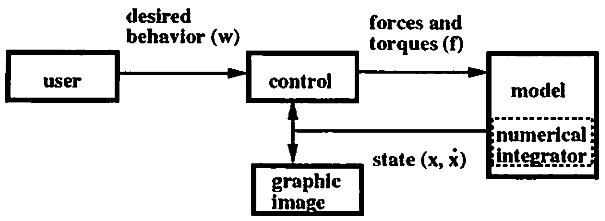


Figure 1: The user specifies the desired behavior at a high level. The control algorithms compute forces and torques that should be applied at each joint based on the state of the system and the desired behavior. The internal forces at the joints and external forces from the environment are applied to the links of the model. The equations of motion are integrated forward in time to produce the new state of the system.

of the number found in humans. The equations of motion for each model were generated with a commercially available package (Rosenthal and Sherman 1986). The package generates efficient subroutines for the equations of motion of the model ($O(n)$ where n is the number of links) using a variant of Kane's method and a symbolic simplification phase.

A state machine provides the underlying structure for the control system for each animation but allows the control laws to change when the dynamics of the system or the user's commands change. For example, the dynamics of the seesaw change when the legs of a rider touch the ground and the control laws must change to match. Similarly, the control laws change when the user switches from controlling the position to controlling the velocity of the unicycle. Each state has specific control laws and the transitions between the states are determined by changes in the state of the system or by changes in the input to the control system.

Proportional-derivative control is used for low-level position control in most of the simulations. The torque exerted at a joint is a function of the error in position and the relative velocity between the links on either side of the joint:

$$\tau = \mathrm{kp}(\theta - \theta_{\mathrm{d}}) + \mathrm{kv}\dot{\theta}$$

where τ is the control torque, θ is the relative joint angle between the links, θ_d is the rest position of the spring, and $\dot{\theta}$ is the relative velocity between the links. The gains, k_p and k_v depend on the mass of the links and the desired stiffness and damping of the joint. This servo has the same effect on the system as a spring and damper where the rest position of the spring is controlled by the control system.

Animations are produced through high-level interactions with the control system. For example, the user specifies the forward speed of the unicycle, how long the juggler should use one pattern before switching to another, or the maximum height the swing will achieve. At each simulation time step, the control system computes forces or torques for each joint based on the state of the system and the requirements of the task. The equations of motion of the system are integrated forward in time, and the resulting motion is displayed in a simple graphical model and recorded for later use in an animation. The layout of the animation system is shown in figure 1.

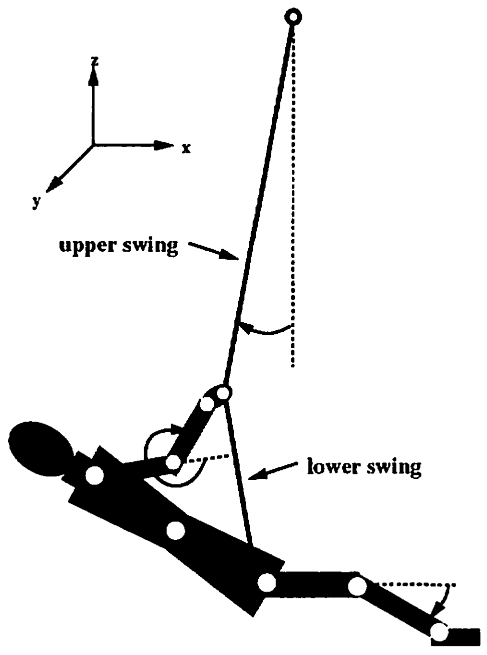


Figure 2: Model used to animate the motion of a human pumping a swing. All joints rotate about the y axis and the motion of the model is constrained to the x-z plane. The angle of the swing, the knee and the elbow are marked to aid in the interpretation of figure 5.

The details of the individual models and control systems are described below.

Pumping a Swing

To pump a swing the control actions of the human must be coordinated with the fore-aft motion of the swing. Figure 2 shows the model used to simulate a human pumping a swing. The links of the model— cylinders, truncated cones, and ellipsoids—are connected by rotary joints at the wrists, elbows, shoulders, waist, hips, knees, and ankles. The swing is modeled as two rods connected by a pivot joint located where the hands attach to the swing. A second pivot joint attaches the bottom of the lower body to the end of the lower swing rod. This joint rotates freely, and the angle between the body and the lower rod of the swing is controlled by the elbow, shoulder, and wrist joints. The grip of the hands on the swing joint is modeled by a pair of orthogonal springs and dampers. The motion of the swing and the human are constrained to the fore-aft plane.

Swinging has been studied for two simple models: a point mass that slides up and down a rigid rod and a system that switches between a double and a compound pendulum (Tea and Falk 1968; Burns 1970; Gore 1970; Gore 1971; McMullen 1972). These models are shown in figure 3. Our model is more complex than either of these, but the simpler models provide some physical intuition about how the control system can move the joints to increase the amplitude of the swinging motion.

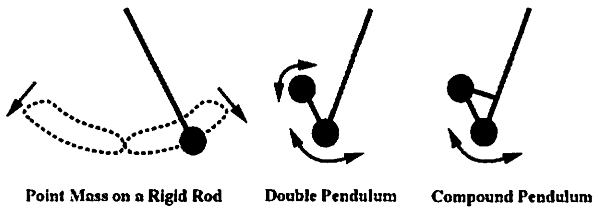


Figure 3: Two simplified models of pumping. The leftmost model, a point mass on a rigid rod, can be pumped by sliding the mass up the rod at the lowest point of the cycle and down at the highest in a pattern like that shown by the dashed line. This pumping action decreases the moment of inertia of the system when all the energy is kinetic and increases it when all the energy is potential. The double/compound pendulum model increases the amplitude of the swing by letting the second pendulum fall backwards at the highest point of the swing. When the fall of the second pendulum is arrested and the model becomes a compound pendulum, the angular velocity of the first pendulum is increased causing the swing to go higher on the next cycle. The motion of a human on a swing is similar to the motion of the double/compound pendulum in that the arms are relaxed at the back of the swing so that the body falls backward and acts like a double pendulum until it is caught by the extended arms. When the arms stop the motion of the body, the system becomes a compound pendulum again but with increased angular velocity.

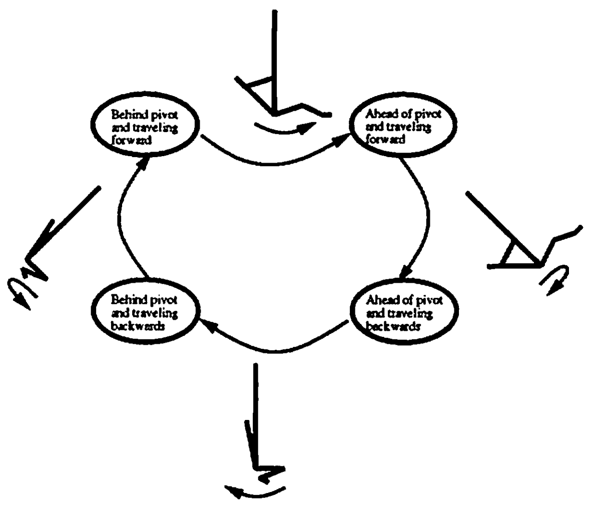

Behind pivot and traveling forward
Ahead of pivot and traveling forward
Behind pivot and traveling backwards
Ahead of pivot and traveling backwards

Figure 4: State machine used to control the pumping motion. The state is determined by the angle and velocity of the swing. The state **behind pivot and traveling forward** has the same control laws as the state **ahead of pivot and traveling forward** and serves only to prevent false transitions before the swing has reached its maximum forward position. For the same reason, **ahead of pivot and traveling backwards** has the same control laws as **behind pivot and traveling backwards.**

The state of the control system for pumping depends on the angle and velocity of the swing. The transitions between the states occur when the swing passes through the lowest point of the cycle and when it nears the highest point. The state machine is illustrated in figure 4. The control laws for each state specify the desired angles and the gains for each servo.

The control system uses a proportional-derivative servo or spring/damper system to control each joint. When the swing is moving forward the desired angles cause the body to lean back and the legs to extend. As the swing moves backwards, the desired angles cause the arms to move the body forward towards the lower rod and the legs to bend. The top graph of figure 5 shows the angle of the swing as the control system pumps for twenty seconds, coasts for ten seconds, and pumps again for ten seconds. The middle graph shows the angle of the knee joint as the legs are swung back and forth. The bottom graph shows the angle of the elbow joint as it pulls the body forward and allows the body to fall backwards.

Riding a Seesaw

The seesaw animation has two riders on opposite ends of a plank. The control system varies the rotation of the plank by changing how hard each of the riders pushes off against the ground. The model for the seesaw animation is shown in figure 6. The collision model for the leg and the ground consists of two orthogonal pairs of springs and dampers. The springs are stretched between the touchdown and current positions of the end of the leg. When the lower end of the leg leaves the ground, the springs are disconnected. Like the hands in the swing model, the lower arms are attached to the handles of the seesaw with springs and dampers.

The control for the seesaw uses two independent state machines, one for each rider. The seesaw with two riders is much like a quadruped bounding in place, and the control algorithms are similar to those used for the control of a bounding quadruped (Raibert and Hodgins 1991). The state machine for the control of the seesaw is shown in figure 7. The transitions between the states occur when the legs touch or leave the ground.

The control laws consist of proportional-derivative servos for each joint. The setpoints and gains depend on the state. During **flight**, the leg is moved to an appropriate position for touchdown. During **compression**, a spring at the knee joint stores energy and causes the system to bounce passively. During **extension**, the knee is extended to add energy to the bouncing oscillation of the system. The height of the oscillation can be varied by changing the extension of the knee.

Juggling

To animate juggling, the control system moves the wrist, shoulder, and elbow joints of a model so that the hands catch and throw balls. The user directs the animation by specifying the juggling pattern (cascade, shower, or fountain) and the length of time that the balls are held in the hand (dwell time) or flying through the air (flight time). The control system perturbs the throws so as to maintain the stability of each pattern and produce transitions between the patterns.

Figure 8 illustrates the model used in the juggling simulation. The hands are not anthropomorphic but are of approximately the same size and density as human hands. Collisions are detected between each ball and the surfaces

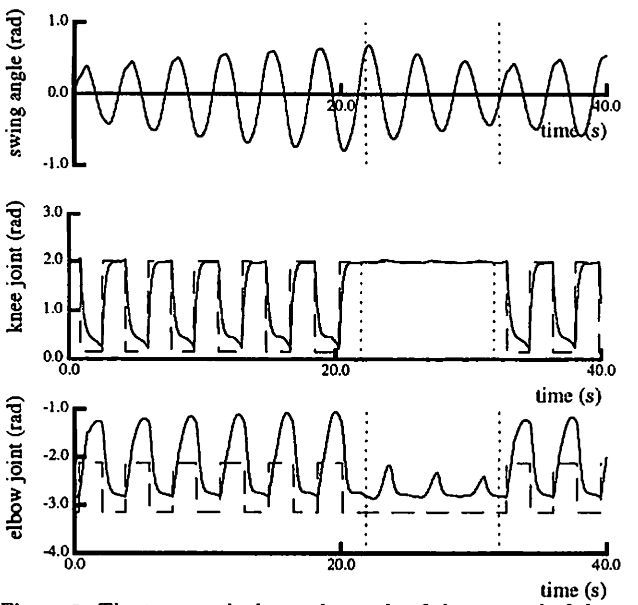


Figure 5: The top graph shows the angle of the top rod of the swing with respect to vertical. The control system pumped the swing for twenty seconds, coasted for ten seconds, and pumped again for ten seconds. The vertical dashed lines indicate when the task changed from **pump** to **coast**. The middle graph shows the movement of the knee joint as the legs are swung back and forth. The bottom graph shows the elbow joint as it pulls the body forward and allows it to fall backwards. The dashed lines in the lower two graphs indicate the desired angle for the knee and elbow. The elbow is pulled away from the desired angle by the weight of the body when the body is leaning back.

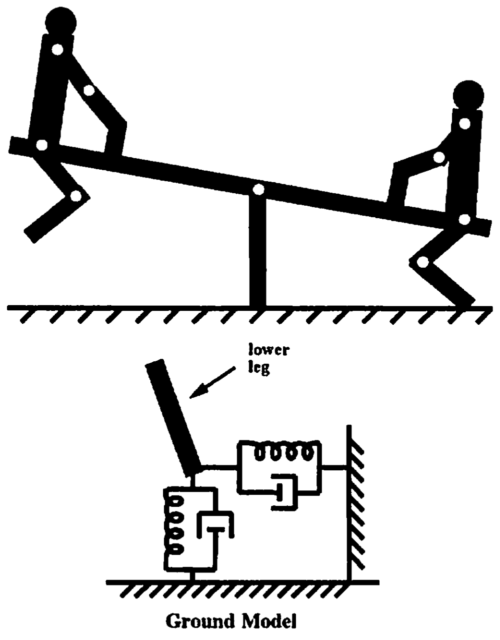


Figure 6: The model for the seesaw animation. The ground model is two orthogonal pairs of springs and dampers. The ends of the arms are also attached to the handles of the seesaw with springs and dampers.

of the fingers, thumbs, and palms and are modeled by a spring normal to the plane of the surface and by two dampers aligned with the surface and perpendicular to each other. Collision forces are applied to both the hand and the ball when the ball is in contact with the hand.

Control System

The basis of all juggling patterns are accurate throws and robust catches. The control system causes the ball to be thrown by generating a desired trajectory for the hand in cartesian coordinates that will accelerate the ball and the hand to the desired velocity by the time they reach the release point. The control system computes torques that will cause the hand to match the speed of the ball as it falls and then close around the ball after contact. This method of catching was implemented for a one degree of freedom juggling robot by Bühler, Koditschek, and Kindlmann (1989).

The control system has four states: **meet**, **decelerate**, **accelerate**, and **follow**. In each state the control system generates a desired trajectory for the hand that will cause it to move to the desired position and arrive with the desired velocity and acceleration. Using inverse kinematics, the desired hand position is transformed to desired positions for the joints of the upper and lower arm and wrist. Proportional-derivative control coupled with a simplified version of the forward dynamics cause each joint to track the trajectory.

During the **meet** state, the hand moves up to meet the falling ball. The control system generates a polynomial trajectory that will cause the hand to meet the ball at the desired catching position with a velocity that matches the velocity of the ball and an acceleration equal to gravity. Matching the descent velocity of the ball reduces the chance that the ball will bounce out of the hand before the finger and thumb close to constrain it.

When the ball is in the hand, the control system is in either the **decelerate** or **accelerate** state. During these states the control system chooses a trajectory that reverses the motion of the hand and ball in the z direction and moves it towards the desired release position. When the hand nears the release position, the finger and thumb open and the hand follows the ball briefly (0.05 sec) in x and y while decelerating in z to prevent collisions which would disturb the trajectory of the ball. After the ball leaves the hand, the control system generates a trajectory that will cause the hand to meet the next ball.

Juggling Patterns

Jugglers commonly use three different three-ball patterns: the cascade, shower, and fountain (Buhler and Graham 1984; Beek 1989). The three patterns are shown in figure 9. In the cascade, each hand throws the balls across to the other hand and each throw passes under the arriving balls. In the shower, the balls move roughly in a circle with the throw from one hand passing above the throw from the other hand. In the fountain, each

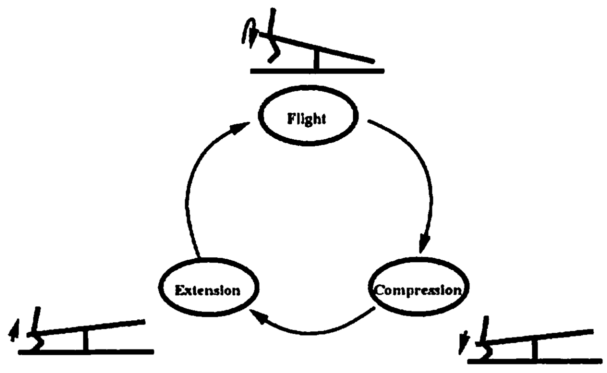


Figure 7: The state machine for one seesaw rider. The transition from **flight** to **compression** occurs when the leg touches the ground. The transition from **compression** to **extension** occurs when the knee joint begins to extend. The transition from **extension** to **flight** occurs when the leg leaves the ground.

hand juggles separately, throwing and catching the balls without passing them to the other hand.

To produce these three juggling patterns we chose release positions, catch positions, and flight times for each pattern. The catching and throwing routines were the same for all patterns.

Transitions between Patterns

The control system performs transitions between patterns by waiting until the first ball in a pattern is caught and then setting the throw and catch positions and the flight times to those used in the new pattern. Two feedback laws make the transitions robust: phase correction and collision avoidance.

The phase of each ball is the time in the cycle at which it is caught. In a three-ball cascade or shower, catching the first ball signals the beginning of a cycle, the second ball is caught 1/3 of the way through the cycle, and the third 2/3 of the way through. Phase correction is performed by adjusting the dwell time so that the next time the ball is caught it will be closer to the correct phase. In the fountain pattern, the two hands operate independently and the phase correction algorithm is used to keep both the balls and the hands operating in phase.

The feedback law for collision avoidance is used when errors in phasing or the changing patterns of the throws would cause two balls to collide. On each throw, the control system uses the ballistic equations for the balls to predict if the ball in the hand will collide with either of the balls in the air. If a collision is expected, the throw position is moved by twice the ball radius in x to prevent the collision.

Balancing on a Unicycle

To maintain balance a unicycle rider must push on the pedals in such a way as to correct errors in balance and forward speed. The user directs the animation by specifying a desired velocity or a desired position.

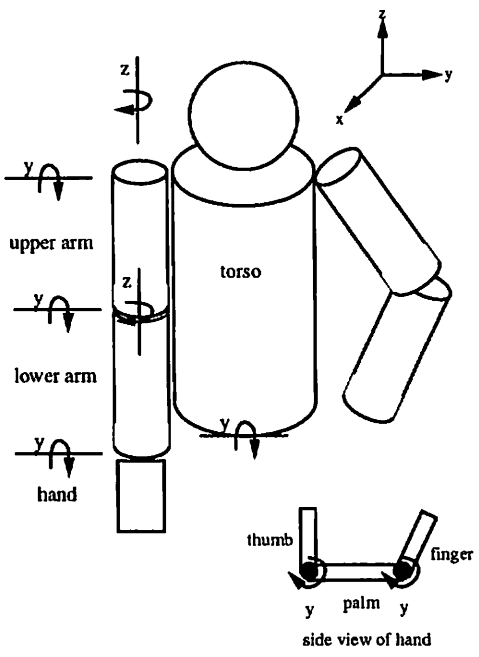


Figure 8: The rigid-body model used in the juggling animation. The model has eleven links with a total of fifteen degrees of freedom. The parameters for the mass, moment of inertia, and dimensions of all the links except the hands were obtained from Dempster and Gaughran (1965). The hands are of approximately the same size and density as the human hand measured in Dempster and Gaughran (1965).

The model contains a wheel, a seat, and a body with two legs (figure 11). The legs push on the pedals through a pair of orthogonal springs and dampers. The springs allow the legs to push down and sideways on the pedals but not to pull up. The contact model between the wheel and the ground is also a pair of springs and dampers. This model allows the wheel to roll but does not allow it to slip.

Control

Unlike the other animations described in this paper, the unicycle control problem is continuous and there are no impacts that cause the dynamics of the system to change. The unicycle wheel is always touching the ground, and the control laws do not depend on the state of the system. As a result, the state machine is used only to handle changes in the control laws when the user switches from position control to velocity control.

The desired torque at the wheel is a function of the error in forward speed and the angle of the stem:

$$\tau = k_\phi \phi + k_{\dot\phi} \dot\phi + k_{\dot x}(\dot x_d - \dot x)$$

where τ is the desired torque at the wheel, ϕ is the angle of the stem relative to vertical, $\dot\phi$ is the velocity of the angle of the stem, $\dot x_d$ is the desired speed of the unicycle, $\dot x$ is the actual speed, and k_ϕ, $k_{\dot\phi}$, and $k_{\dot x}$ are gains.

There is no motor at the hub, and the desired torque at the wheel is produced by moving the hip and knee

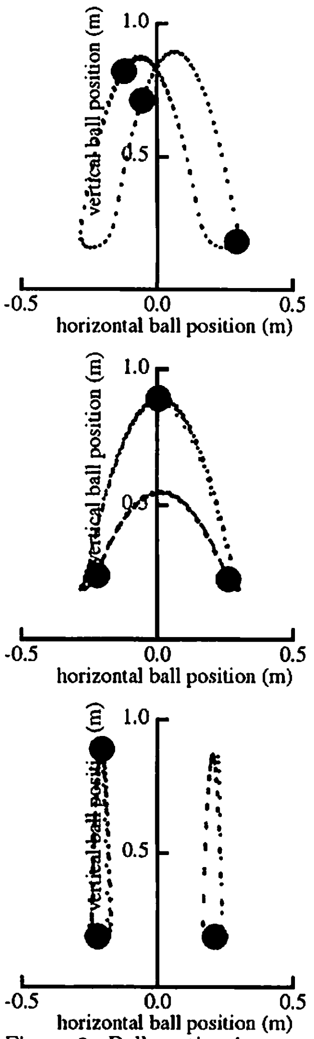


Figure 9: Ball motion in cascade, shower, and fountain patterns.

joints so that the legs press down on the pedals with the appropriate force. Each leg is assigned a percentage of the desired torque based on the sign of the desired torque and the current pedal position:

$$\text{weight} = \frac{1}{2} + \frac{1}{2}\sin\theta$$

The other leg has a weighting of 1 − weight. When the torque is positive and the pedals are horizontal ($\theta = \pm\pi/2$) the front leg has a weighting of one and the rear leg has a weighting of zero. When the pedals are vertical, both legs have a weighting of one-half and the force is divided evenly between the two legs. The desired force along the axis of the leg is increased by a preload to ensure that the legs always push down on the pedals and remain on the pedals.

The desired force at the pedal is converted to desired torques at the knee and hip by taking the kinematics of

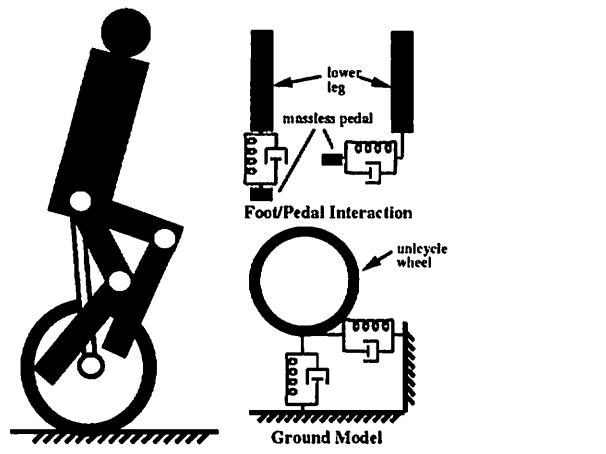


Figure 11: Schematic drawing of the model used in the unicycle animation. The model consists of a wheel, a seat, and a body with two legs divided into upper and lower segments. The body is attached to the seat by a pivot joint. The interaction between the lower legs and the massless pedals is modeled by a pair of orthogonal spring/dampers. The springs allow the legs to push down and sideways on the pedals but not to pull up. The ground model for the unicycle wheel is a pair of orthogonal spring/dampers. The x spring is stretched between the point marking the distance that the wheel has rolled and the point on the ground where the wheel is touching. The point to which the wheel has rolled is $x_0 + 2\pi r\theta$ where x_0 is the starting location, r is the radius of the wheel, and θ is the number of revolutions of the wheel. The point at which the wheel touches the ground is assumed to be the point directly beneath the hub. This model does not allow the wheel to slip.

the leg into account and assuming massless legs:

$$\tau_{knee} = -\text{ll} f_x$$

$$\tau_{hip} = -\text{ul}(\sin(\alpha) f_z + \cos(\alpha) f_x) - \text{ll} f_x$$

where τ_{knee} is the desired torque at the knee, τ_{hip} is the desired torque at the hip, ll is the length of the lower leg, ul is the length of the upper leg, α is the angle of the knee, and f_x and f_z are the desired forces between the pedal and the lower leg in the coordinate system of the lower leg.

The user interacts with the animation of a unicycle rider by specifying a desired speed or a desired position. The desired position is converted to a desired speed:

$$\dot{x}_d = k_x(x - x_d)$$

where $\dot{x}_d$ is the desired speed, x is the current position, x_d is the desired position, and k_x is a gain. The desired speed is limited by a maximum desired speed. Figure 12 shows the behavior of the system as it starts from rest and cycles to a specified location.

The control for the unicycle simulation has constant gains. Vos (Vos 1989; Vos and von Flotow 1990) states that the gains must be adjusted based on the state of the system for good performance of a three-dimensional robot unicycle with a motor at the hub and a horizontal turntable for yaw control. When the simulation is extended to three dimensions and the operating range of

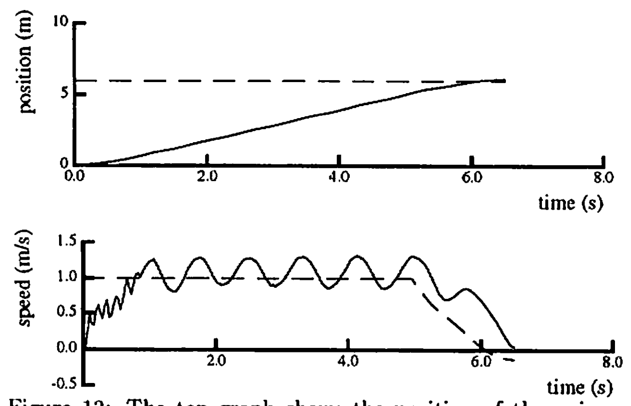


Figure 12: The top graph shows the position of the unicycle as it moves towards the goal. The bottom graph shows the forward speed. The dashed lines indicate the desired position and velocity.

the simulation is extended, we may also find this to be true but constant gains provide adequate performance for the two-dimensional model.

Discussion

We have presented descriptions of the models and control systems for four animations: a swing, a seesaw, a juggler, and a unicycle. In each case, the control system is built upon a state machine and the transitions between the states are determined by changes in the state of the system or changes in the input commands. The current state determines which control laws should be used to control the system. The interactions with the environment and many of the low-level control laws are implemented with springs and dampers. The state machines and some of the control laws are hand designed for each animation.

If properly designed, a control system will produce natural-looking motion when it is used to activate a physically realistic model. The combination of a physically realistic model and a control system can easily produce bad motion too, by violating joint or torque limits or by performing the task in an unexpected way or with extraneous motions. This problem is particularly apparent in underconstrained problems like the swing. There are many ways to pump a swing but only a small fraction produce motion that resembles the movements made by a human pumping a swing.

Our experiments during the design of these animations suggest that several features are required for the generation of natural-looking motion:

- The model must contain the key features of the system being modeled. If the model is too simple, the motion may not appear natural. We originally modeled the swing as a single rigid rod (no joint where the hand is attached) and discovered that the setpoints had to be adjusted very carefully for the swinging motion to increase in amplitude. With the extra degree of freedom, the control is much more robust and the motion appears more natural. We also compared a unicycle animation that was powered by the motion of the legs with one in which the control was provided by a torque source at the wheel and the legs were positioned kinematically. The resulting motions differed because the motor could exert a uniform torque in all pedal positions while the legs could not.
- The model should include the control parameters. In designing and tuning a control system, we found it easy to set the gains too high so that the simulation produced jerky motion. A strength model should be part of the physical model and control commands that require the physical system to go beyond those limits should be filtered. Lee et al (1990) implemented such a system for the task of lifting a load.
- Biological data can provide information about setpoints and control actions. In an underconstrained problem like the swing, choosing the control actions is not easy. A wide range of actions cause the swing to gain amplitude but only a much smaller range are commonly used by humans on a swing. We studied measurements of people pumping a swing to learn about the joint angles that were used and the transition points of the control system. More rigorous comparison of simulation data with biological data can also be used to show to what extent the simulated motion resembles the biological motion.
- Perfectly smooth motion is not natural. Steady-state simulations often produce motion that is too repetitive. Each of the phase plots of juggling shown in figure 9 contains data from several repetitions of the pattern, but there is little variation from one toss to the next. This too-perfect motion appears "robot-like" when played back through a graphical model. We need to take advantage of all opportunities for adding interest to the motion: changes in high-level commands from the user, disturbances from the environment, and the addition of noise within the system.

We have not addressed the question of automatic generation of control systems. Large portions of each of the control systems described here were designed by hand with the aid of principles gleaned from our experience in controlling robots. We are interested in exploring other approaches to the problem of generating control systems, perhaps by formalizing the principles that were outlined here to allow the automatic generation of control systems for limited domains or by applying techniques from optimal control theory to design the control systems with less input from the human designer.

By approaching the problem of motion generation from the perspective of physical realism, we have gained rules that make it possible to automatically generate motion through simulation. Within this framework we can generate many different motions that accomplish a single task by varying the physical system and the control strategies. For example, an adult pumps a swing differently than the eight-year-old child we modeled and a skilled unicyclist would use different gains and perhaps even different control laws than a beginner. Even greater variety could be produced by violating physical laws. In hand-drawn animations emotion and humor are often communicated through exaggerated motions that violate physical laws. Eventually, we would like to take advantage of this part of the design space; however, we feel that a better understanding of techniques for the generation of physically correct motion is needed first.

Acknowledgments

We thank Marc Raibert and the MIT Leglab for loaning us part of the simulation environment and Dave Haumann and Alan Norton for their enthusiastic support of this project.

References

Baraff, D. 1989. Analytical methods for dynamic simulation of non-penetrating rigid bodies. *Computer Graphics*, 23(3): 223-232.

Baraff, D. 1991. Coping with Friction for Non-penetrating Rigid Body Simulation. *Computer Graphics* 25(4):31-40.

Barzel, R., Barr, A. H. 1988. A Modeling System Based on Dynamic Constraints. *Computer Graphics* 22(4):179-188.

Beek, P. J., 1989. *Juggling Dynamics.* (Free University Press: Amsterdam).

Bruderlin, A., Calvert, T. W. 1989. Goal-Directed, Dynamic Animation of Human Walking. *Computer Graphics* 23(3):233-242.

Bühler, M., Koditschek, D. E., Kindlmann, P. J. 1989. Planning and Control of Robotic Juggling Tasks. In *Robotics Research: The Fifth International Symposium.* (MIT Press: Cambridge).

Buhler, J., Graham, R. 1984 Fountains, Showers, and Cascades. *The Sciences*, 24:44-51.

Burns, J. A. 1970. More on Pumping a Swing. *American Journal of Physics* 38:920-922.

Dempster, W. T., Gaughran, G. R. L. 1965. Properties of Body Segments based on Size and Weight. *American Journal of Anatomy* 120: 33-54.

Gore, B. F. 1970. The Child's Swing. *American Journal of Physics* 38:378-379.

Gore, B. F. 1971. Starting a Swing from Rest. *American Journal of Physics* 39:347.

Hahn, J. 1988. Realistic Animation of Rigid Bodies. *Computer Graphics* 22(4):299-308.

Hodgins, J. K. 1991. Biped Gait Transitions. In *Proceedings of the IEEE International Conference on Robotics and Automation*, Sacramento, CA.

Hodgins, J., Raibert, M. H. 1990. Biped Gymnastics. *International Journal of Robotics Research*, 9(2):115-132

Kass, M., Miller, G. 1990. Rapid, Stable Fluid Dynamics for Computer Graphics. *Computer Graphics* 24(4):49-57.

Lee, P., Wei, S., Zhao, J., Badler, N.I. 1990. Strength Guided Motion. *Computer Graphics* 24(4):253-262.

McKenna, M., Zeltzer, D. 1990. Dynamic Simulation of Autonomous Legged Locomotion. *Computer Graphics* 24(4):29-38.

McMullen, J. T. 1972. On Initiating Motion in a Swing. *American Journal of Physics* 40;764-766.

Meredith, H. V. 1969a. Body Size of Contemporary Groups of Eight-year-old Children Studied in Different Parts of the World. Monographs of the Society for Research in Child Development 34(1).

Meredith, H. V. 1969b. Body Size of Contemporary Youth in Different Parts of the World. Monographs of the Society for Research in Child Development 34(7).

Miller, G. S. P., 1988. The Motion Dynamics of Snakes and Worms. *Computer Graphics* 22(4):169-178.

Norton, A., Turk, G., Bacon, R., Gerth, J., Sweeney, P. 1991. Animation and Fracture by Physical Modeling. *The Visual Computer*, 7(4):210-219.

Pentland, A., Williams, J. 1989. Good Vibrations: Modal Dynamics for Graphics and Animation. *Computer Graphics* 23(3):215-222.

Raibert, M. H. 1986. Legged Robots that Balance. (MIT Press: Cambridge).

Raibert, M. H., Hodgins, J. K, 1991. Animation of Dynamic Legged Locomotion. *Computer Graphics* 25(4):349-358.

Rosenthal, D. E., Sherman, M. A., 1986. High performance multibody simulations via symbolic equation manipulation and Kane's method. *J. Astronautical Sciences* 34(3):223-239.

Tea, P. L., Jr., Falk, H. 1968. Pumping on a Swing. *American Journal of Physics* 36:1165-1166.

Terzopoulos, D., Witkin, A. 1988. Physically Based Models with Rigid and Deformable Components. *IEEE Computer Graphics and Applications* 8(6).

Terzopoulos, D., Fleischer, K. 1988. Modeling Inelastic Deformation: Viscoelasticity, Plasticity, Fracture. *Computer Graphics* 22(4), 269-278.

van de Panne M., Fiume, E., Vranesic, Z. 1990. Reusable Motion Synthesis Using State-Space Controllers *Computer Graphics* 24(4):225-234.

Vos, D. W., von Flotow, A. H. 1990. Dynamics and Nonlinear Adaptive Control of an Autonomous Unicycle: Theory and Experiment. Proceedings of the 29th Conference on Decision and Control. Honolulu, Hawaii, 182-187.

Vos, D. W. 1989. Dynamics and Nonlinear, Adaptive Control of an Autonomous Unicycle. M.S. Thesis, Department of Aeronautics and Astronautics, Massachusetts Institute of Technology.

Wejchert, J., Haumann, D. 1991. Animation Aerodynamics. *Computer Graphics* 25(4):19-22.

Witkin, A., Kass, M., 1988. Spacetime Constraints. *Computer Graphics* 22(4):159-168.

Beyond Keyframing: An Algorithmic Approach to Animation

A. James Stewart
Department of Computer Science
University of Toronto

James F. Cremer
Computer Science Department
Cornell University

Abstract

The use of physical system simulation has led to realistic animation of passive objects, such as sliding blocks or bouncing balls. However, complex *active* objects like human figures and insects need a control mechanism to direct their movements. We present a paradigm that combines the advantages of both physical simulation and algorithmic specification of movement. The animator writes an *algorithm* to control the object and runs this algorithm on a physical simulator to produce the animation. Algorithms can be reused or combined to produce complex sequences of movements, eliminating the need for keyframing. We have applied this paradigm to control a biped which can walk and can climb stairs. The walking algorithm is presented along with the results from testing with the *Newton* simulation system.

CR categories: I.3.7 [Computer Graphics]: Three dimensional graphics and realism - animation; I.6.3 [Simulation and Modeling]: Applications

Keywords: physical simulation, human figure animation, animation control, constraints, dynamics

Introduction

This paper describes a paradigm for the control and animation of complex active objects such as the human figure. In this approach the animator develops an *algorithm* which controls the object by specifying certain intuitive variables as a function of time and of world state. The algorithm is able to continuously monitor the world state as it is being automatically updated by an underlying dynamics simulation system, and the algorithm is able to react when it sees changes in the world state.

For example, in the case of biped walking, the animator might write an algorithm that controls the angle of the knees at one point in the animation, and that controls the trajectory of the foot at another point. The algorithm might monitor the world state and, when it notes an event such as a foot touching the ground, stop controlling the trajectory of the heel and start controlling the angle of the knee.

Most animators would probably be comfortable with the idea of "programming" a human figure to walk. The algorithmic approach to animation allows this to be done with ease. This is demonstrated by the walking algorithm presented below.

Other Dynamics Work in Graphics

In some of the first work in this area, **Armstrong and Green** [1] present the equations of motion for tree-structured linkages of rigid bodies and discuss an efficient method of solving them.

Witkin and Kass [27] have combined physical simulation and keyframing to produce realistic animation of their jumping Luxo lamp. With their approach the animator uses *spacetime constraints* to specify several key points for selected variables at specific times. Combining spacetime constraint equations with the Lagrangian equations of motion and discretizing over time yields a system of equations that are solved to produce the motion. Our algorithmic approach differs in that the constraints can be added or removed "on the fly" as the algorithm sees changes in the world state which might not be predictable.

Herr and Wyvill [15] describe a dynamics simulation system which allows easy user control through a simulation language and several high level control primitives. Our work is similar in that the user can define and control arbitrary variables, but we concentrate more on developing algorithms to control complex objects in an intuitive manner.

van de Panne, Fiume, and Vranesic [25] build state-space controllers to provide control torques that achieve desired goal states from arbitrary initial states. Such controllers can be concatenated to produce movement, including cyclic movement like walking.

Other approaches to combine control and physical simulation have been explored: **Wilhelms** [26] and **Barzel and Barr** [3] blend kinematic and dynamic analysis, **Moore and Wilhelms** [22] and **Baraff** [2] discuss the collision and contact problems, **Isaacs and Cohen** [18] incorporate inverse dynamics in their simulation system, and **Brotman and Netravali** [5] use dynamics and optimal control to interpolate between key frames.

Other Work in Walking and Control

The algorithmic approach is meant as a general method by which to control complex mechanisms. In this paper we use the walking problem as an example of an application of the algorithmic approach. Some other approaches to walking are briefly described here.

Kearney, Hansen, and Cremer [19], in an approach very similar to ours, examine the control of mechanical systems as a constraint programming problem. **Bruderlin and Calvert** [6] have developed an effective goal directed approach to dynamic walking in which the animator specifies a few high-level walking parameters. **McKenna and Zeltzer** [20] develop a gait controller and low-level motor programs to generate legged motion. **Zeltzer** [28] analyzes various approaches to the control of complex animated objects and considers their integration. **Raibert and Hodgins** [24] describe control systems for several legged creatures. **Brooks** [4] produces complex walking behavior in a physical, insect-like robot from a distributed network of low-level finite state machines.

Other Work in Robotics

Some further insights on control can be gained from examining the literature in the field of robotics. While this field deals with controlling real, physical objects, some of the techniques can be applied to animation.

Researchers in robotics have taken various approaches to reduce the complexity of control programs for physical objects. The computed torque method for robot arms (see **Craig** [8]) can be viewed as simplifying control by reducing the gripper to a unit mass. The control program can ignore the dynamics of the robot arm, only concerning itself with the position of the end effector as a function of time.

In building his one-legged hopping machine, **Raibert** [23] partitioned control along three intuitive degrees of freedom: hopping, forward speed and body posture. This resulted in surprisingly simple control programs for the hopping robot. For multi-legged machines, Raibert introduced the idea of a "virtual leg" which was defined in terms of the robot's physical legs. This again led to simplified control programs.

Both the computed torque method and Raibert's virtual leg demonstrate that a proper choice of control variables can lead to simplified control programs. The problem with this approach is that there is often no simple closed-form mapping of these control variables onto the forces and torques needed to control the object. In some cases a complete system of equations must be numerically solved to make this mapping. This is called "inverse dynamics" and is typically rejected by robotics researchers as being too expensive to use in real-time control. For the purposes of animation, however, it is ideal. Our application of inverse dynamics will be described in the next section.

The Algorithmic Approach

In the algorithmic approach, the animator's algorithm selects a small set of intuitive variables with which to control the object over the course of the simulation. The algorithm can control predefined variables, such as the forces and torques at the joints, or the instantaneous translational and rotational accelerations of the various components of the object. The algorithm can *also* control variables that it defines as linear combinations of these predefined variables.

For example, the algorithm could, with the appropriate subroutine call to the underlying simulation system, define the acceleration of the center of mass of an object as

$$a_{cm} = \frac{1}{M}\sum_i m_i \cdot a_i,$$

where m_i is the mass of the i^{th} component of the object (a constant), and a_i is the translational acceleration of the i^{th} component. Then at each time step of the simulation, the algorithm could supply a value for a_{cm}.

The underlying simulation system, called *Newton*, is a general purpose physical simulator. Given a description of a complex object (in, say, a computer file), *Newton* will automatically generate the corresponding system of Newton-Euler equations of motion which describe the instantaneous behavior of the object. *Newton* can then integrate these equations of motion over time to produce the animation. *Newton* also automatically updates the system of equations as kinematic relationships in the simulation change (one such change would occur as the biped's foot touches the ground).

The animator's algorithm interacts with Newton in the following ways:

- The algorithm can add arbitrary equations and variables to *Newton*'s system of motion equations. In the example above, the algorithm added a variable, a_{cm}, and a equation defining that variable in terms of other variables of the system. The algorithm can remove equations that it previously added to the system of motion equations.
- The algorithm can set the value of a variable at any time step of the simulation. In the example above, the algorithm could supply a value for the a_{cm} variable at each time step.
- It may be that the algorithm manipulates *Newton*'s system of motion equations such the system becomes underconstrained, admitting many solutions. In this event, the algorithm can tell *Newton*, through an appropriate subroutine call, to select a motion that instantaneously minimizes some quadratic function of the variables of the system.
- The algorithm can observe the world state and act upon it. For example, a walking algorithm might observe that the heel has touched the ground and react by moving into a new state of its execution (like a finite-state machine).

At each time step of the simulation, *Newton* evaluates the current system of equations to determine values for any unknown variables, including the translational and rotational accelerations of the individual components of the object. *Newton* integrates these accelerations to produce the state at the next time step, and this process is iterated.

Format of a Control Algorithm

A control algorithm can be considered as a set of finite state machines. Each machine has an initial state and a transition between states is made when some user-defined predicate become true.[1]

In the algorithm of Figure 1 there is a single machine having initial state START and having one transition START -> CM-ACCEL. The transition is made immediately, and defines a new unknown variable, $\ddot{r}_{cm}$, causes an equation[2] to be added to the system of motion equations, defines a function f which will be called whenever *Newton* needs a value for $\ddot{r}_{cm}$, and defines a quadratic minimization function. Note that the object which is being simulated must be defined elsewhere.

```
initial-states { START }

transition START -> CM-ACCEL when TRUE

  begin
  new-unknown " $\ddot{r}_{cm}$ "
  add-equation " $\ddot{r}_{cm} = \frac{1}{M}\sum m_i \ddot{r}_i$ "
  add-function " $\ddot{r}_{cm} = f($ time $)$ "
  add-quadratic " $Q = \sum \ddot{r}_i + \sum \dot{\omega}_i$ "
  end
```

Figure 1: A Simple Control Algorithm

Overview of Newton

The walking algorithm described in this paper has been designed and tested using the *Newton* simulation system [9] developed at Cornell University. The development of *Newton* was inspired by the need for more general-purpose, flexible simulation systems.

Extensive mechanical engineering research has led to many developments in physical system simulation. The ADAMS [7] and DADS [14] systems are examples of large state-of-the-art systems from the mechanical engineering domain. Such systems are sophisticated in many ways: they support efficient formulations of mechanism dynamics, they use fancy numerical techniques for solving equation systems, they often handle object flexibility and elasticity, and so on. The recent work by graphics and animation researchers (discussed above) has generally been less sophisticated but has placed greater emphasis on animation of interesting high-degree-of-freedom mechanisms.

Still, none of these systems combines the full range of features required to make dynamics simulation as powerful and useful as it could be. Typically they have almost ignored geometric considerations and represented objects simply as point masses with associated inertias and coordinate systems. Geometric modeling techniques have matured enough to allow object representations used by dynamic simulations to include a complete geometric description usable by a geometry processing module. Furthermore, impact, contact, and friction are typically handled by current systems in an *ad hoc* or rudimentary manner, if at all. In some cases, for instance, any possible impacts must be specified in advance; in others, a kind of "force field" technique is used, in which between every pair of objects there is a repelling force that is negligible except when objects are very close together. In addition, the desire to manipulate high-degree-of-freedom objects suggests that a module for specification of control algorithms should be a significant part of a dynamics system.

Newton Architecture

Using *Newton*, a designer can define complex three-dimensional physical objects and mechanisms and can represent object characteristics from various domains. An object consists of a number of "models," each responsible for organization of object characteristics from a particular domain. In most simulations the basic domains of geometry, dynamics, and controlled behavior are modeled. A dynamic modeling system, for example, is responsible for maintaining an object's position, velocity, and acceleration, and for automatically formulating the object's dynamics equations of motion. A geometric modeling system is responsible for information about an object's shape, distinguished features on the object, and computation of geometric integral properties such as volume and moments of inertia. It also detects and analyzes object interpenetrations so that an interference modeling system can deal with collisions between objects.

Newton has three main components: the definition and representation module, the analysis module and the report system. The definition module analyzes high level language descriptions of *Newton* entities and organizes the corresponding data structures. The analysis component implements the top-level control loop of simulations and coordinates the working of various analysis subsystems. The report system handles generation of graphical feedback to users during simulations as well as recording of relevant information for later regeneration of animations.

[1] For the sake of clarity the algorithms will be described in a Pascal-like notation (however, they are currently written in Lisp).

[2] We use quotation marks to indicate that the actual equations must be represented in some internal manner.

Dynamic Analysis in Newton

A physical object is modeled as a collection of rigid bodies related by constraints. Newton-Euler equations of motion are associated with each individual rigid body. At the time an object is created the equations are of the form

$$m\ddot{r} = 0$$
$$J\dot{\omega} + \omega \times J\omega = 0,$$

where m is the mass, $\ddot{r}$ is the second time derivative of the position (i.e. the acceleration), J is the 3×3 inertia matrix, and ω and $\dot{\omega}$ are the rotational velocity and acceleration, respectively.

A specification that two objects are to be connected with a spherical hinge is met by the addition of one vectorial constraint equation and the addition of some terms to the motion equations of the constrained objects. For a holonomic constraint such as this one, the second derivative of the constraint equation can be used along with the modified motion equations to solve for object accelerations and reaction forces. Thus, the equations above become

$$\begin{aligned} m_1\ddot{r}_1 &= F_{hinge} \\ J_1\dot{\omega}_1 + \omega_1 \times J_1\omega_1 &= c_1 \times F_{hinge} \\ m_2\ddot{r}_2 &= -F_{hinge} \\ J_2\dot{\omega}_2 + \omega_2 \times J_2\omega_2 &= c_2 \times -F_{hinge} \end{aligned}$$

$$\ddot{r}_1 + \dot{\omega}_1 \times c_1 + \omega_1 \times (\omega_1 \times c_1) = \ddot{r}_2 + \dot{\omega}_2 \times c_2 + \omega_2 \times (\omega_2 \times c_2),$$

where c_i is the vector from object i's center of mass to the location of the hinge and F_{hinge} is the constraint force that keeps the objects together. Note that the last equation above is the second time derivative of the holonomic constraint equation $r_1 + c_1 = r_2 + c_2$ for spherical joints. Other kinds of "hinges" commonly used in *Newton* include revolute or pin joints, prismatic joints, springs and dampers, and rolling contacts.

If gravity is present during the simulation the system will automatically add gravitational force terms to the objects' translational motion equations. The system keeps track of the constraints responsible for the various terms in the motion equations. Thus, constraints, and their corresponding motion equation terms, can be removed at any time without necessitating complete rederivation of the system of motion equations.

Using this method of dynamics formulation, closed-loop kinematic chains are handled as simply as open chains. Though the formulation does lead to a large set of equations, the matrices are very sparse and often symmetric. Thus, acceptable efficiency is achieved by the use of sparse matrix solution techniques.

Event handling, impact and contact

Newton, unlike many other simulation systems, can automatically and incrementally reformulate the motion equations as *exceptional events* occur during simulations.

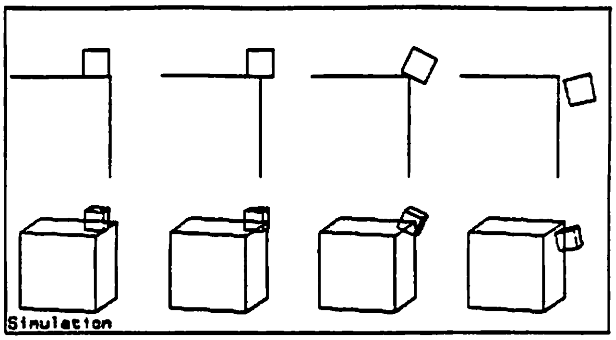


Figure 2: Changing Kinematic Relationships

One kind of exceptional event is a change in kinematic relationship between objects. Figure 2 shows a block that was initially sliding along a table top. After some time the edge of the table is reached and the contact relationship changes from a plane-plane contact to a plane-edge contact. Still later the contact is broken altogether. These changing contact relationships are automatically detected by *Newton*. The system of motion equations and the related constraint equations are automatically maintained by *Newton* to reflect these changing relationships.

Newton's event handler is primarily responsible for detection and resolution of impacts, for analysis of continuous contacts between objects, for the corresponding maintenance of *temporary hinges* that model unilateral constraints between objects in contact, and for handling of events specified by control programs that necessitate changes in the constraint set. For example, the walking algorithm might tell the event handler to notify it when the biped's foot touches the ground so that it can change the constraint equations.

The geometric modeling subsystem is responsible for detecting and analyzing impacts and interpenetrations. In the usual method of handling impacts, the dynamic analysis module formulates impulse-momentum equations in a manner completely analogous to the formulation of the basic dynamics equations, and solves these equations to produce the instantaneous velocity changes caused by the impact. The details of *Newton*'s methods for handling impact, contact and other exceptional events are given elsewhere [16, 17, 11, 10].

Event definition and control

Support for control programming is provided by allowing users to define their own event types. Events provide the mechanism for state transitions in control programs. Event definition consists of a specification of how to detect the event (including information about how accurately the time of event occurrence should be isolated) and how to resolve it.

```
procedure position-with-PD( equation-name, object,
                            x-desired, delta-time )

  var $x$, $v$, $a$:  quantity
      $\tau$:  real

  begin
  $x$ = get-position-quantity( object )
  $v$ = get-velocity-quantity( object )
  $a$ = get-acceleration-quantity( object )

  $\tau$ = - delta-time / log( .01 )

  add-named-equation( equation-name,
          " $a + \frac{2}{\tau} v + \frac{1}{\tau^2}(x -$ x-desired$) = 0$ " )
  end
```

Figure 3: PD Controller Used in Positioning

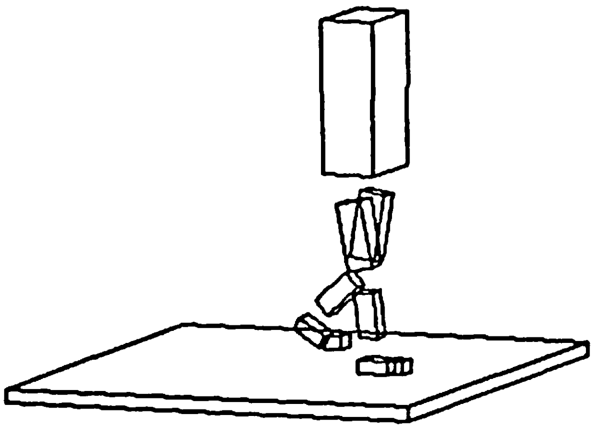


Figure 4: Simulated Biped Model

Low-level Controllers

In designing algorithms with *Newton* we found ourselves frequently using PD (proportional-derivative) controllers and curve-fitting controllers to control the "trajectory" of many of the defined quantities. In controlling the biped, for example, quintic interpolation was used to plot the trajectory of the heel, and a PD controller was used to orient the foot before it struck the ground. A small library of these controllers is used in the biped algorithm, and will be described here.

PD controllers are used in the biped algorithm to control orientation, position and joint angle. Each controller adds an equation to the system of motion equations which defines the second derivative of the quantity in terms of the first derivative and the quantity itself.

The procedure in Figure 3 adds an equation which produces accelerations to move an object to within 1% of a position x-desired within a given time delta-time. The equation continues to affect the object's motion until it is explicitly removed by the control algorithm. The quantities x, v and a are data structures representing state variables of the controlled object. These data structures are used by the add-named-equation function to create the appropriate equation.

The Biped Algorithms

We have developed two algorithms to control a biped: one for straight-line walking and one for walking up stairs. An abbreviated version of the walking algorithm is shown in Figure 8.

The simulated biped consists of a torso, two legs with knee joints and two feet with toe joints. This model was adapted from a description of McMahon [21] and is shown in Figure 4. The hips are three degree of freedom spherical joints, the ankles are two degree of freedom universal joints, while the knees and toes are one degree of freedom revolute joints, making a total of fourteen degrees of freedom. The biped is about 180 centimeters tall, weighs 85 kilograms, and has moments approximating those of a human being.

Walking Algorithm

For ease of exposition, the walking algorithm of Figure 8 is an abbreviated version of our actual algorithm. We have hidden many of the lower level procedures (in particular, those which compute the trajectory of the heel). The actual algorithm is written is Lisp; a simulation language like that of Herr and Wyvill [15] will be implemented in the future.

The algorithm has three states: START, SWING and DOUBLE-SUPPORT. Consider the START -> SWING transition in Figure 8. After this transition (that is, during the SWING phase) the torso is forced to remain in a fixed orientation by the TORSO-ORIENTATION constraint. The swing foot follows a trajectory defined by an equation called SWING-HEEL-TRAJECTORY which was determined by the procedure move-heel-to-target, the stance leg is stiffened with set-angle-with-PD, the foot is oriented for landing with orient-with-PD, and the angle of the toe is set with set-angle-with-PD.

In the DOUBLE-SUPPORT phase, the constraints on the swing foot are removed, the names of the swing and stance legs are swapped, and the torso is constrained to accelerate slightly forward, which helps the trailing heel to lift.

The largest number of constraints are applied in the SWING phase, during which eleven scalar equations have been added to *Newton's* system of motion equations. Since the biped has fourteen degrees of freedom, it remains underconstrained at all times. A quadratic cost function Q is defined in order to fully determine the motion of the biped (a motion is chosen to minimize Q). The cost function is a weighted sum of the translational

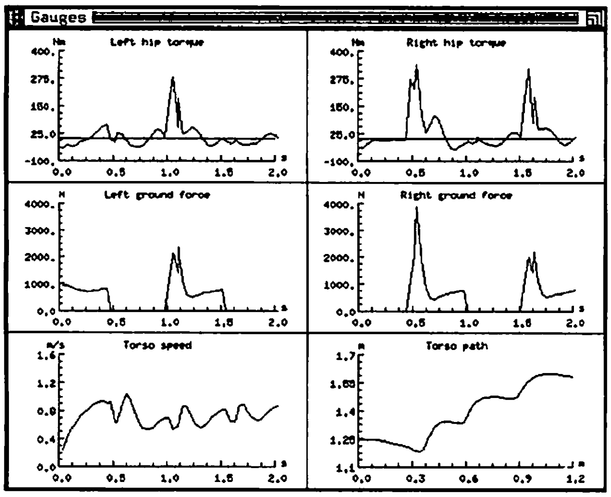


Figure 5: *Newton* Statistical Output

and angular accelerations, and of the difference between the torso translational acceleration and some acceleration defined by a function **F** which tries to keep the torso mid-way between the two feet.

A slightly more complex walking algorithm was actually implemented and tested with the *Newton* simul: tion system. Figure 6 shows ten frames in which the biped completes a full cycle. The full simulation consisted of twenty seconds of straight-line walking on a flat surface.

Stair Climbing

Another version of the algorithm was developed for stair climbing. The principal differences between the walking and climbing algorithms were: a more complicated function to determine the trajectory (it has to avoid the steps), a "loose constraint" holding the torso upright, which allowed the torso to sway in a natural manner (this is explained below), and various parameter changes (for example, the foot strike orientation will be different when climbing stairs than when walking).

Figure 7 shows six frames (side view and back view) in which the biped lifts the right foot. Note that the torso sways slightly (the degree of sway can be changed trivially) and that the torso moves from side to side to be over the supporting foot.

Discussion

The walking algorithm of Figure 8 looks almost too simple to be true. While a lot of the underlying procedures have not been described in this paper, the real reason for this simplicity is that *Newton* automatically handles almost all of the underlying dynamics, and, if we choose, can also automatically handle the detection and resolution of impact and contact.[3]

Due to the simplicity of our current biped model, the algorithms are forced to use too many constraints to achieve the desired motion. In particular, the trajectory of the heel must be exactly specified, yielding motion which can sometimes appear unnaturally stiff. Experiments have shown that the best way to avoid this stiffness is to "loosely constrain" the heel trajectory by adding a weighted term to the minimization function **Q**. This weighted term is the square of the difference between the actual toe acceleration and a computed acceleration which guides the toe along the desired trajectory.

Future Work

We will experiment with elastic tendons in the hope that the swing phase will not have to specify an explicit trajectory for the heel. Instead, no torque would be applied in the swing leg; it would be pulled forward by the stored energy of the stretched tendons. This might approximate "ballistic walking" as described by McMahon[21].

The algorithms will be extended to include downstairs walking and turning on a level surface. Once a suite of such algorithms has been developed, we will be able to define a set of high level commands such as "walk forward" and "step up". With these commands, the animation of walking bipeds should be a simple task for the animator.

Summary

We have presented an algorithmic approach to control. This approach allows the animator to choose intuitive degrees of freedom by which to control an object. The control algorithm adds and removes constraint equations "on the fly" as the world state changes; *a priori* knowledge of the exact moment of each state change is not required.

With the algorithmic approach, all consideration of dynamics and impact is left to the *Newton* simulation system. The burden on the animator is further reduced by allowing underdetermined specification of motion through the use of constrained optimization techniques.

We have presented an algorithm to control a simulated biped, along with results from its execution on the *Newton* simulation system. The algorithm has the advantage of being intuitive, simple to program, and reusable.

[3]For the sake of efficiency, two additional finite state machines — one for each foot — are used to deal with impact and contact, rather than allowing *Newton* to do so in a more general, and hence more expensive, manner. These finite state machines are hidden from the animator.

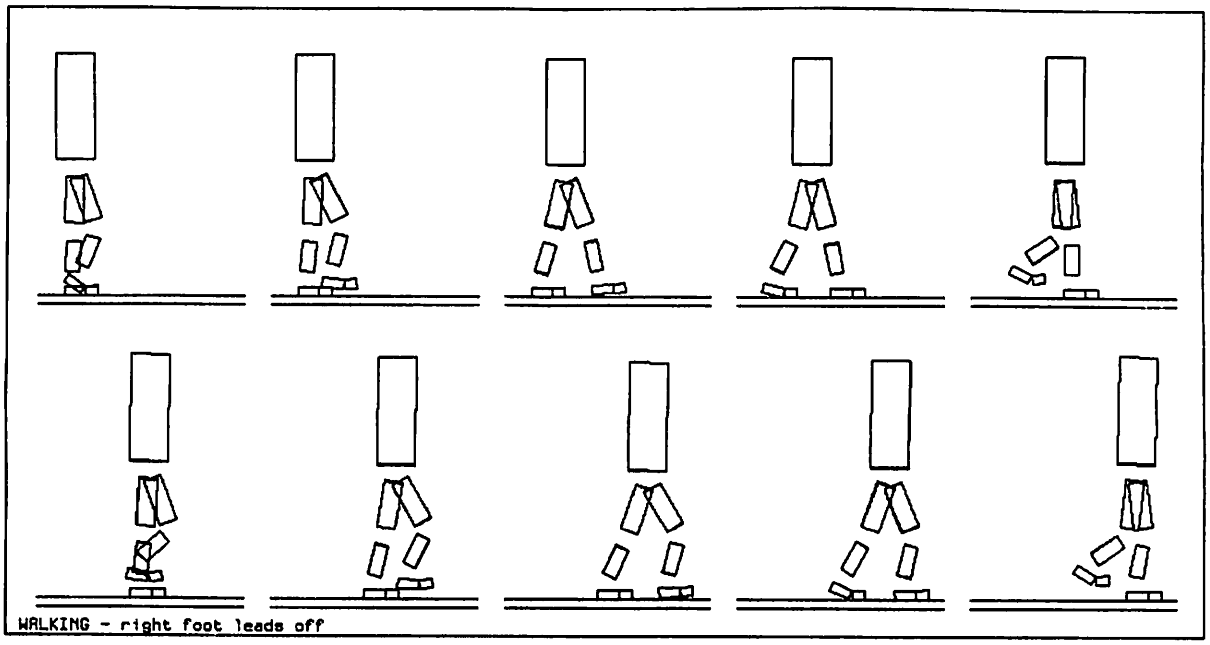


Figure 6: Walking Cycle

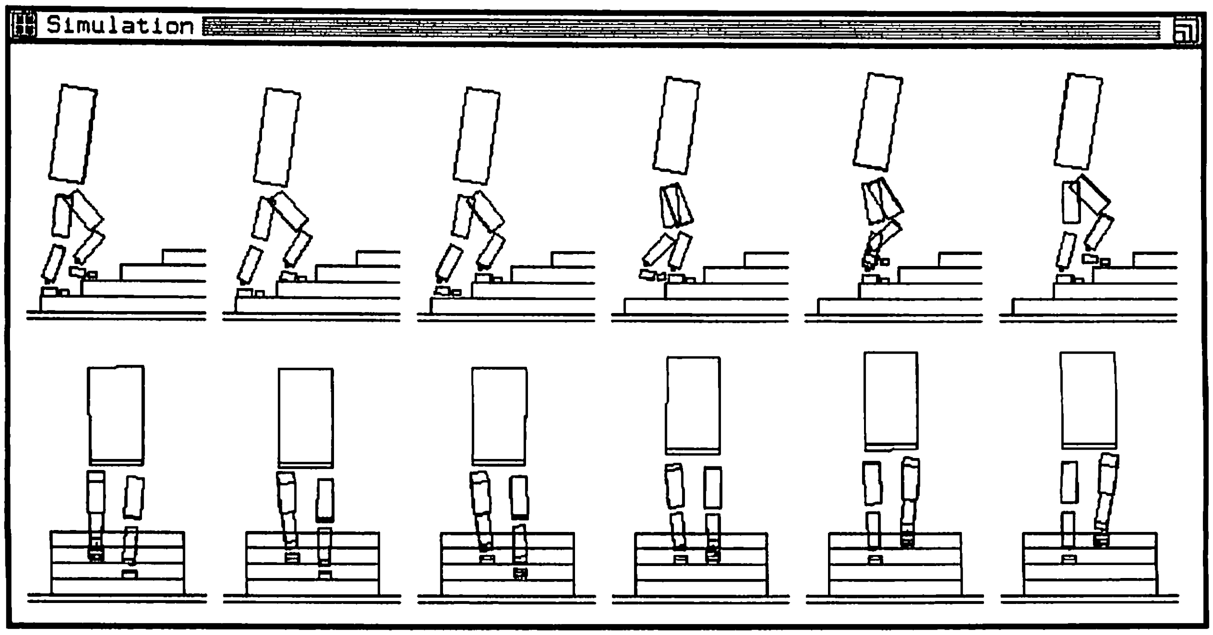


Figure 7: Climbing Cycle

```
const   time-in-air               = 0.5 s
        foot-strike-orientation   = 10° about (0 0 1)
        torso-orientation         = -10° about (0 0 1)


let F = $K_p\ (r_{torso} - \frac{1}{2}(r_{lfoot} + r_{rfoot}))\ +\ K_v\ (\dot{r}_{torso} - \frac{1}{2}(\dot{r}_{lfoot} + \dot{r}_{rfoot}))$

let Q = $(\sum\dot{\omega}^2\ +\ \sum\ddot{r}^2\ +\ 20\,(\ddot{r}_{torso}-\mathbf{F})^2)$


initial-states = { START }


transition START -> SWING when TRUE

  begin
  add-quadratic( Q )
  orient-with-PD(      TORSO-ORIENTATION,      TORSO, torso-orientation, .2 s )
  move-heel-to-target( SWING-HEEL-TRAJECTORY,  swing-heel )
  set-angle-with-PD(   STANCE-KNEE-ANGLE,      stance-knee, 175°, 0.1 s )
  orient-with-PD(      SWING-FOOT-ORIENTATION, swing-foot,  foot-strike-orientation, time-in-air )
  set-angle-with-PD(   SWING-TOE-ANGLE,        swing-toe,   0°, time-in-air )
  end

transition SWING -> DOUBLE-SUPPORT when hits-ground( swing-foot )

  begin
  remove-equations( SWING-HEEL-TRAJECTORY, SWING-FOOT-ORIENTATION, SWING-TOE-ANGLE )
  swap-swing-and-stance()
  accelerate-torso( TORSO-ACCELERATION )
  end

transition DOUBLE-SUPPORT -> SWING when leaves-ground( swing-foot )

  begin
  remove-equation( TORSO-ACCELERATION )
  remove-equation( STANCE-KNEE-ANGLE )
  move-heel-to-target( SWING-HEEL-TRAJECTORY,  swing-heel )
  set-angle-with-PD(   STANCE-KNEE-ANGLE,      stance-knee, 175°, 0.1 s )
  orient-with-PD(      SWING-FOOT-ORIENTATION, swing-foot,  foot-strike-orientation, time-in-air )
  set-angle-with-PD(   SWING-TOE-ANGLE,        swing-toe,   0°, time-in-air )
  end
```

Figure 8: Abbreviated Walking Algorithm

References

[1] W. W. Armstrong and M. W. Green. The dynamics of articulated rigid bodies for purposes of animation. *The Visual Computer*, 1:231–240, 1985.

[2] D. E. Baraff. Analytical methods for dynamic simulation of non-penetrating rigid bodies. In *Computer Graphics (SIGGRAPH 89)*, pages 223–231, 1989.

[3] R. Barzel and A. H. Barr. A modeling system based on dynamic constraints. In *Computer Graphics (SIGGRAPH 88)*, pages 179–188. ACM, August 1988.

[4] R. A. Brooks. A robot that walks: emergent behaviors from a carefully evolved network. In *Proceedings of the 1989 IEEE International Conference on Robotics and Automation*, pages 692–694c, May 1989.

[5] L. S. Brotman and A. N. Netravali. Motion interpolation by optimal control. In *Computer Graphics (SIGGRAPH 88)*, pages 309–315. ACM, August 1988.

[6] A. Bruderlin and T. W. Calvert. Goal-directed, dynamic animation of human walking. In *Computer Graphics (SIGGRAPH 89)*, pages 233–242, 1989.

[7] M. Chace. Modeling of dynamic mechanical systems. Presented at the CAD/CAM Robotics and Automation Institute and International Conference, Tuscon, Arizona, February 1985.

[8] J. J. Craig. *Introduction to Robotics: Mechanics and Control.* Addison Wesley, 1986.

[9] J. F. Cremer. *An Architecture for General Purpose Physical System Simulation — Integrating Geometry, Dynamics, and Control.* PhD thesis, Cornell University, May 1989.

[10] J. F. Cremer. *An architecture for general purpose physical system simulation - Integrating geometry, dynamics, and control.* PhD thesis, Cornell University, 1989. also as Cornell technical report TR 89-987.

[11] J. F. Cremer and A. J. Stewart. Using the *newton* simulation system as a testbed for control. In *Proceedings of the 3rd IEEE International Symposium on Intelligent Control*, 1988.

[12] R. Featherstone. The dynamics of rigid body systems with multiple concurrent contacts. In O. D. Faugeras and G. Giralt, editors, *Robotics Research: The Third International Symposium*, pages 191–196. The MIT Press, 1985.

[13] J. K. Hahn. Realistic animation of rigid bodies. In *Computer Graphics (SIGGRAPH 88)*, pages 299–308. ACM, August 1988.

[14] E. J. Haug and G. M. Lance. Developments in dynamic system simulation and design optimization in the center for computer aided design: 1980-1986. technical report 87-2, University of Iowa, February 1987.

[15] C. Herr and B. Wyvill. Towards generalised motion dynamics for animation. In *Graphics Interface*, pages 49–59, 1990.

[16] C. M. Hoffmann and J. E. Hopcroft. Simulation of physical systems from geometric models. *IEEE Journal of Robotics and Automation*, RA-3(3):194–206, June 1987.

[17] C. M. Hoffmann, J. E. Hopcroft, and M. S. Karasick. Towards implementing robust geometric computations. In *ACM Annual Symposium on Computational Geometry*, pages 106–117, June 1988.

[18] P. M. Isaacs and M. F. Cohen. Controlling dynamic simulation with kinematic constraints, behavior constraints and inverse dynamics. In *Computer Graphics (SIGGRAPH 87)*, pages 215–224. ACM, July 1987.

[19] J. K. Kearney, S. Hansen, and J. F. Cremer. Programming mechanical simulations. In *Proceedings of the 2nd Eurographics Workshop on Animation and Simulation*, pages 223–243, September 1991.

[20] M. McKenna and D. Zeltzer. Dynamic simulation of autonomous legged locomotion. In *Computer Graphics (SIGGRAPH 90)*, pages 29–38, 1990.

[21] T. A. McMahon. Mechanics of locomotion. *The International Journal of Robotics Research*, 3(2):4–28, 1984.

[22] M. Moore and J. Wilhelms. Collision detection and response for computer animation. In *Computer Graphics (SIGGRAPH 88)*, pages 289–298. ACM, August 1988.

[23] M. H. Raibert. *Legged Robots That Balance.* The MIT Press, 1986.

[24] M. H. Raibert and J. K. Hodgins. Animation of dynamic legged locomotion. In *Computer Graphics (SIGGRAPH 91)*, pages 349–358, 1991.

[25] M. van de Panne, E. Fiume, and Z. Vranesic. Reusable motion synthesis using state-space controllers. In *Computer Graphics (SIGGRAPH 90)*, pages 225–234, 1990.

[26] J. Wilhelms. Using dynamic analysis for realistic animation of articulated figures. *IEEE Computer Graphics and Applications*, 7(6):12–27, 1987.

[27] A. Witkin and M. Kass. Spacetime constraints. In *Computer Graphics (SIGGRAPH 88)*, pages 159–168. ACM, August 1988.

[28] D. Zeltzer. Towards an integated view of 3-d animation. *The Visual Computer*, 1:245–259, June 1985.

A Minimalist Global User Interface[1]

Rob Pike
AT&T Bell Laboratories
Murray Hill, New Jersey 07974
`rob@research.att.com`

Abstract

`Help` is a combination of editor, window system, shell, and user interface that provides a novel environment for the construction of textual applications such as browsers, debuggers, mailers, and so on. It combines an extremely lean user interface with some automatic heuristics and defaults to achieve significant effects with minimal mouse and keyboard activity. The user interface is driven by a file-oriented programming interface that may be controlled from programs or even shell scripts. By taking care of user interface issues in a central utility, `help` simplifies the job of programming applications that make use of a bitmap display and mouse.

Keywords: Windows, User Interfaces, Minimalism

Background

Ten years ago, the best generally available interface to a computer was a 24×80 character terminal with cursor addressing. In its place today is a machine with a high-resolution screen, a mouse, and a multi-window graphical user interface. That interface is essentially the same whether it is running on a PC or a high-end 3D graphics workstation. It is also almost exactly the same as what was available on the earliest bitmap graphics displays.

The decade that moved menus and windows from the research lab to more than ten million PC's, that changed computer graphics from an esoteric specialty to a commonplace, has barely advanced the state of the art in user interfaces. A case can be made that the state of the art is even backsliding: the hardware and software resources required to support an X terminal are embarrassing, yet the text editor of choice in universities on such terminals continues to be a character-based editor such as `vi` or `emacs`, both holdovers from the 1970's. With the exception of the Macintosh, whose users have found many creative ways to avoid being restrained (or insulted) by the decision that they would find more than one mouse button confusing, the new generation of machines has not freed its users from the keyboard-heavy user interfaces that preceded them.

There are many reasons for this failure — one that is often overlooked is how uncomfortable most commercially made mice are to use — but the most important might well be that the interfaces the machines offer are just not very good. Spottily integrated and weighed down by layers of software that provide features too numerous to catalog and too specialized to be helpful, a modern window system expends its energy trying to look good, either on a brochure or on a display. What matters much more to a user interface is that it *feel* good. It should be dynamic and responsive, efficient and invisible [Pike88]; instead, a session with X windows sometimes feels like a telephone conversation by satellite.

Where will we be ten years from now? CRT's will be a thing of the past, multimedia will no longer be a buzzword, pen-based and voice input will be everywhere, and university students will still be editing with `emacs`. Pens and touch-screens are too low-bandwidth for real interaction; voice will probably also turn out to be inadequate. (Anyway, who would want to work in an environment surrounded by people talking to their computers?) Mice are sure to be with us a while longer, so we should learn how to use them well.

With these churlish thoughts in mind, I began a couple of years ago to build a system, called `help`, that would have as efficient and seamless a user interface as possible. I deliberately cast aside all my old models of how interfaces should work: the goal was to learn if I could do better. I also erased the usual divisions between components: rather than building an application or an editor or a window system, I wanted something that centralized a very good user interface and made it uniformly available to all the components of a system.

Introduction

`Help` is an experimental program that combines aspects of window systems, shells, and editors to address these issues in the context of textual applications. It is designed to support software development, but falls short of being a true programming environment. It is not a 'toolkit'; it

[1]This is a revision of a paper by the same title published in the Proceedings of the Summer 1991 USENIX Conference, Nashville, 1991, pp. 267-279.

is a self-contained program, more like a shell than a library, that joins users and applications. From the perspective of the application (compiler, browser, etc.), it provides a universal communication mechanism, based on familiar Unix® file operations, that permits small applications — even shell procedures — to exploit the graphical user interface of the system and communicate with each other. For the user, the interface is extremely spare, consisting only of text, scroll bars, one simple kind of window, and a unique function for each mouse button — no widgets, no icons, not even pop-up menus. Despite these limitations, `help` is an effective environment in which to work and, particularly, to program.

The inspiration for `help` comes from Wirth's and Gutknecht's Oberon system [Wirt89, Reis91]. Oberon is an attempt to extract the salient features of Xerox's Cedar environment and implement them in a system of manageable size. It is based on a module language, also called Oberon, and integrates an operating system, editor, window system, and compiler into a uniform environment. Its user interface is disarmingly simple: by using the mouse to point at text on the display, one indicates what subroutine in the system to execute next. In a normal Unix shell, one types the name of a file to execute; instead in Oberon one selects with a particular button of the mouse a module and subroutine within that module, such as `Edit.Open` to open a file for editing. Almost the entire interface follows from this simple idea.

The user interface of `help` is in turn an attempt to adapt the user interface of Oberon from its language-oriented structure on a single-process system to a file-oriented multiprocess system, Plan 9 [Pike90]. That adaptation must not only remove from the user interface any specifics of the underlying language; it must provide a way to bind the text on the display to commands that can operate on it: Oberon passes a character pointer; `help` needs a more general method because the information must pass between processes. The method chosen uses the standard currency in Plan 9: files and file servers.

The interface seen by the user

This section explains the basics of the user interface; the following section uses this as the background to a major example that illustrates the design and gives a feeling for the system in action.

`Help` operates only on text; at the moment it has no support for graphical output. A three-button mouse and keyboard provide the interface to the system. The fundamental operations are to type text with the keyboard and to control the screen and execute commands with the mouse buttons. Text may be selected with the left and middle mouse buttons. The middle button selects text defining the action to be executed; the left selects the object of that action. The right button controls the placement of windows. Note that typing does not execute commands; newline is just a character.

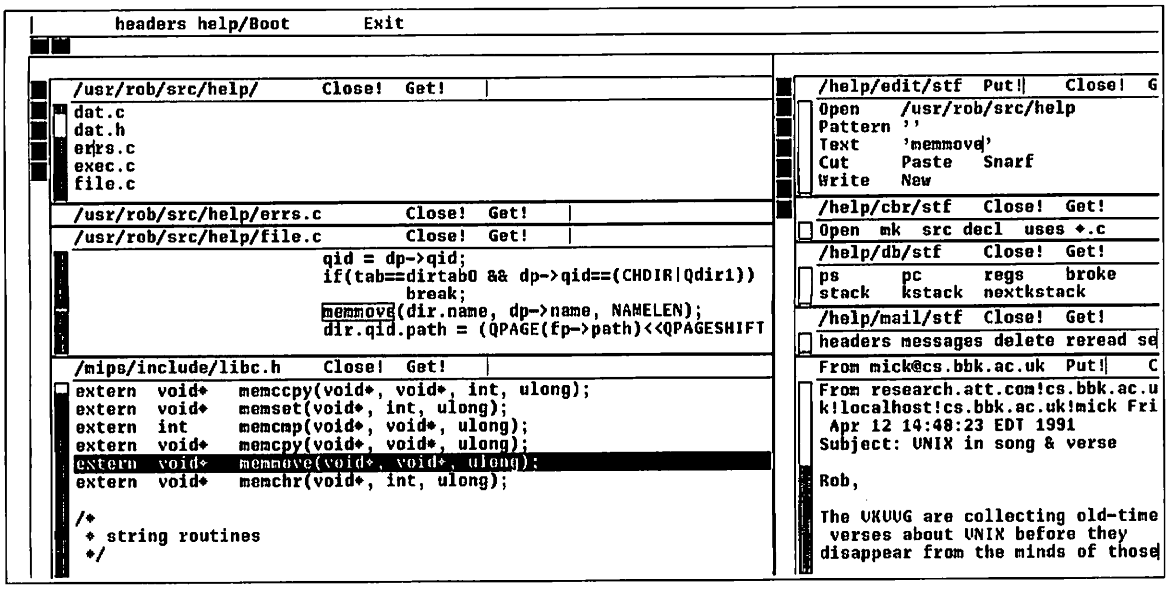
Several interrelated rules were followed in the design of the interface. These rules are intended to make the system as efficient and comfortable as possible for its *users*. First, *brevity:* there should be no actions in the interface — button clicks or other gestures — that do not directly affect the system. Thus `help` is not a 'click-to-type' system because that click is wasted; there are no pop-up menus because the gesture required to make them appear is wasted; and so on. Second, *no retyping:* it should never be necessary or even worthwhile to retype text that is already on the screen. (Many systems allow the user to copy the text on the screen to the input stream, but for small pieces of text such as file names it often seems easier to retype the text than to use the mouse to pick it up, which indicates that the interface has failed.) As a corollary, when browsing or debugging, rather than just typing new text, it should be possible to work efficiently and comfortably without using the keyboard at all. Third, *automation:* let the machine fill in the details and make mundane decisions. For example, it should be good enough just to point at a file name, rather than to pass the mouse over the entire textual string. Finally, *defaults:* the most common use of a feature should be the default. Similarly, the smallest action should do the most useful thing. Complex actions should be required only rarely and when the task is unusually difficult.

The `help` screen is tiled with windows of editable text, arranged in (usually) two side-by-side columns. Figure 1 shows a `help` screen in mid-session. Each window has two subwindows, a single *tag* line across the top and a *body* of text. The tag typically contains the name of the file whose text appears in the body.

The text in each subwindow (tag or body) may be edited using a simple cut-and-paste editor integrated into the system. The left mouse button selects text; the selection is that text between the point where the button is pressed and where it is released. Each subwindow has its own selection. One subwindow — the one with the most recent selection or typed text — is the location of the *current selection* and its selection appears in reverse video. The selection in other subwindows appears in outline.

Typed text replaces the selection in the subwindow under the mouse. The right mouse button is used to rearrange windows. The user points at the tag of a window, presses the right button, drags the window to where it is desired, and releases the button. `Help` then does whatever local rearrangement is necessary to drop the window to its new location (the rule of automation). This may involve covering up some windows or adjusting the position of the moved window or other windows. `Help` attempts to make at least the tag of a window fully visible; if this is impossible, it covers the window completely.

A tower of small black squares, one per window, adorns the left edge of each column. (See Figure 1.) These tabs represent the windows in the column, visible or invisible, in order from top to bottom of the column, and can be clicked with the left mouse button to make the corresponding window fully visible, from the tag to the bottom of the column it is in. A similar row across the top of the columns allows the columns to expand horizontally. These little tabs are an ade-



```
headers help/Boot        Exit

/usr/rob/src/help/        Close!  Get!
dat.c
dat.h
errs.c
exec.c
file.c
/usr/rob/src/help/errs.c             Close!  Get!
/usr/rob/src/help/file.c             Close!  Get!
                        qid = dp->qid;
                        if(tab==dirtab0 && dp->qid==(CHDIR|Qdir1))
                                break;
                        memmove(dir.name, dp->name, NAMELEN);
                        dir.qid.path = (QPAGE(fp->path)<<QPAGESHIFT
/mips/include/libc.h      Close!  Get!
extern  void*   memccpy(void*, void*, int, ulong);
extern  void*   memset(void*, int, ulong);
extern  int     memcmp(void*, void*, ulong);
extern  void*   memcpy(void*, void*, ulong);
extern  void*   memmove(void*, void*, ulong);
extern  void*   memchr(void*, int, ulong);

/*
 * string routines
 */
```

```
/help/edit/stf  Put!    Close!  G
Open    /usr/rob/src/help
Pattern ''
Text    'memmove'
Cut     Paste   Snarf
Write   New
/help/cbr/stf   Close!  Get!
Open  mk  src decl  uses *.c
/help/db/stf    Close!  Get!
ps      pc      regs    broke
stack   kstack  nextkstack
/help/mail/stf  Close!  Get!
headers messages delete reread se
From mick@cs.bbk.ac.uk  Put!    C
From research.att.com!cs.bbk.ac.u
k!localhost!cs.bbk.ac.uk!mick Fri
 Apr 12 14:48:23 EDT 1991
Subject: UNIX in song & verse

Rob,

The UKUUG are collecting old-time
 verses about UNIX before they
disappear from the minds of those
```

Figure 1: A small `help` screen showing two columns of windows. The current selection is the black line in the bottom left window. The directory `/usr/rob/src/help` has been `Opened` and, from there, the source files `/usr/rob/src/help/errs.c` and `file.c`.

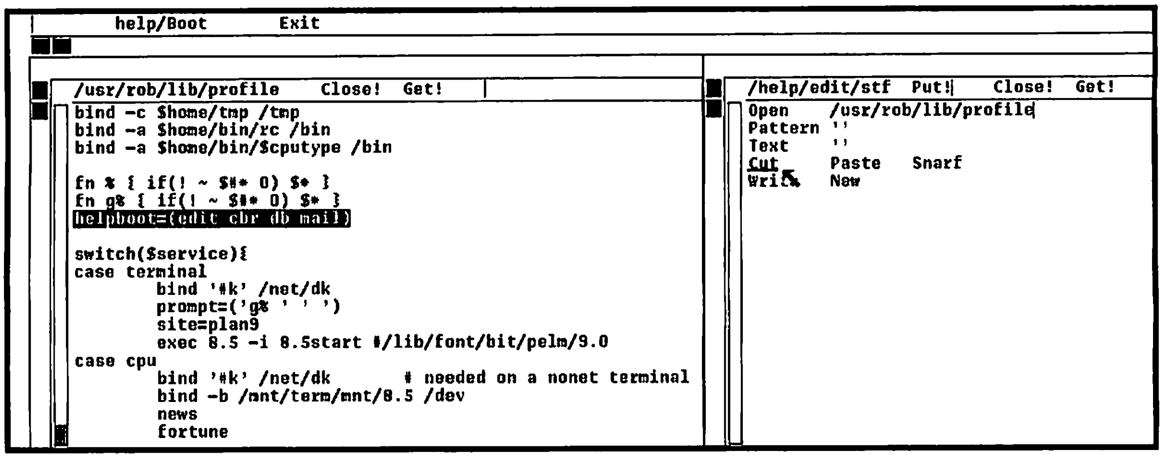
quate but not especially successful solution to the problem of managing many overlapping windows. The problem needs more work; perhaps the file name of each window should pop up alongside the tabs when the mouse is nearby.

Like the left mouse button, the middle button also selects text, but the act of releasing the button does not leave the text selected; rather it executes the command indicated by that text. For example, to cut some text from the screen, one selects the text with the left button, then selects with the middle button the word `Cut` anywhere it appears on the display. (By convention, capitalized commands represent built-in functions.) As in any cut-and-paste editor, the cut text is remembered in a buffer and may be pasted into the text elsewhere. If the text of the command name is not on the display, one just types it and *then* executes it by selecting with the middle button. Note that `Cut` is not a 'button' in the usual window system sense: it is just a word, wherever it appears, that is bound to some action. To make things easier, `help` interprets a middle mouse button click (not *double* click) anywhere in a word as a selection of the whole word (the rule of defaults). Thus one may just select the text normally, then click on `Cut` with the middle button, involving less mouse activity than with a typical pop-up menu. If the text for selection or execution is the null string, `help` invokes automatic actions to expand it to a file name or similar context-dependent block of text. If the selection is non-null, it is always taken literally.

As an extra acceleration, `help` has two commands invoked by chorded mouse buttons. While the left button is still held down after a selection, clicking the middle button executes `Cut`; clicking the right button executes `Paste`, replacing the selected text by the contents of the cut buffer. These are the most common editing commands and it is convenient not to move the mouse to execute them (the rules of brevity and defaults). One may even click the middle and then right buttons, while holding the left down, to execute a cut-and-paste, that is, to remember the text in the cut buffer for later pasting.

More than one word may be selected for execution; executing `Open /usr/rob/lib/profile` creates a new window and puts the contents of the file in it. (If the file is already open, the command just guarantees that its window is visible.) Again, by the rule of automation, the new window's location will be chosen by `help`. The hope is to do something sensible with a minimum of fuss rather than just the right thing with user intervention. This policy was a deliberate and distinct break with most previous systems. (It is present in Oberon and in most tiling window systems but `help` takes it farther.) This is a contentious point, but `help` is an experimental system. One indication that the policy is sound is that minor changes to the heuristics often result in dramatic improvements to the feel of the system as a whole. With a little more work, it should be possible to build a system that feels just right.

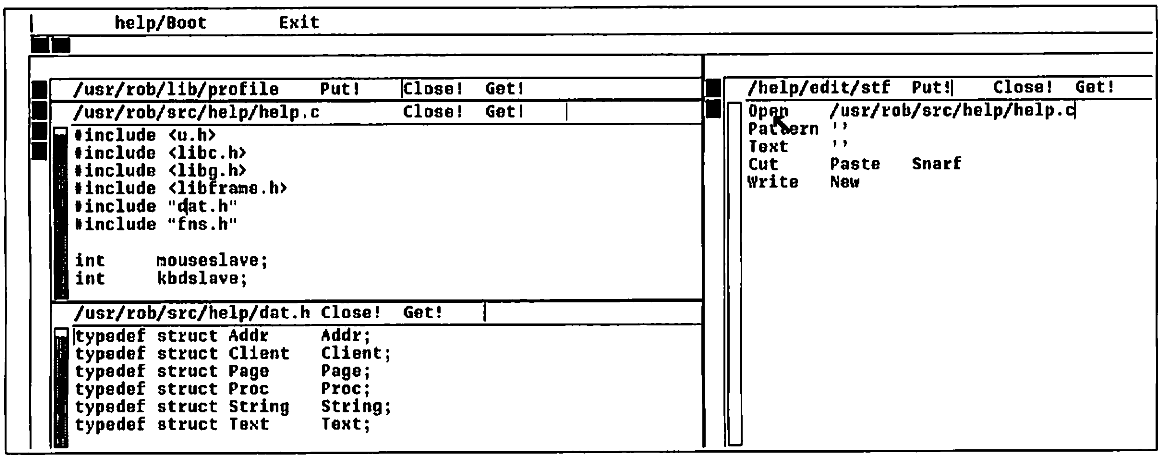
A typical shell window in a traditional window system permits text to be copied from the typescript and presented as input to the shell to achieve some sort of history function: the ability to re-execute a previous command. `Help` instead tries to predict the future: to get to the screen commands and text that will be useful later. Every piece of text on the screen is a potential command or argument for a command. Many of the basic commands pull text to the screen from the file system



```
help/Boot        Exit

/usr/rob/lib/profile    Close!  Get!
bind -c $home/tmp /tmp
bind -a $home/bin/rc /bin
bind -a $home/bin/$cputype /bin

fn % { if(! ~ $#* 0) $* }
fn g% { if(! ~ $#* 0) $* }
helpboot=(edit cbr db mail)

switch($service){
case terminal
        bind '#k' /net/dk
        prompt=('g% ' ' ')
        site=plan9
        exec 8.5 -i 8.5start #/lib/font/bit/pelm/9.0
case cpu
        bind '#k' /net/dk        # needed on a nonet terminal
        bind -b /mnt/term/mnt/8.5 /dev
        news
        fortune
```

```
/help/edit/stf  Put!    Close!  Get!
Open     /usr/rob/lib/profile
Pattern  ''
Text     ''
Cut      Paste    Snarf
Write    New
```

Figure 2: Executing Cut by sweeping the word while holding down the middle mouse button. The text being selected for execution is underlined.



```
help/Boot        Exit

/usr/rob/lib/profile    Put!    Close!  Get!
/usr/rob/src/help/help.c        Close!  Get!
#include <u.h>
#include <libc.h>
#include <libg.h>
#include <libframe.h>
#include "dat.h"
#include "fns.h"

int     mouseslave;
int     kbdslave;

/usr/rob/src/help/dat.h Close!  Get!
typedef struct Addr     Addr;
typedef struct Client   Client;
typedef struct Page     Page;
typedef struct Proc     Proc;
typedef struct String   String;
typedef struct Text     Text;
```

```
/help/edit/stf  Put!    Close!  Get!
Open     /usr/rob/src/help/help.c
Pattern  ''
Text     ''
Cut      Paste    Snarf
Write    New
```

Figure 3: Opening files. After typing the full path name of help.c, the selection is automatically the null string at the end of the file name, so just click Open to open that file; the defaults grab the whole name. Next, after pointing into dat.h, Open will get /usr/rob/src/help/dat.h.

with a minimum of fuss. For example, if Open is executed without an argument, it uses the file name containing the most recent selection (the rule of defaults). Thus one may just point with the left button at a file name and then with the middle button at Open to edit a new file. Using all four of the rules above, if Open is applied to a null selection in a file name that does not begin with a slash (/), the directory name is extracted from the file name in the tag of the window and prepended to the selected file name. An elegant use of this is in the handling of directories. When a directory is Opened, help puts the its name, including a final slash, in the tag and just lists the contents in the body. (See Figure 1.)

For example, by pointing at dat.h in the source file /usr/rob/src/help/help.c and executing Open, a new window is created containing the contents of /usr/rob/src/help/dat.h: two button clicks. (See Figure 3.) Making any non-null selection disables all such automatic actions: the resulting text is then exactly what is selected.

That Open prepends the directory name gives each window a context: the directory in which the file resides. The various commands, built-in and external, that operate on files derive the directory in which to execute from the tag line of the window. Help has no explicit notion of current working directory; each command operates in the directory appropriate to its operands.

The Open command has a further nuance: if the file name is suffixed by a colon and an integer, for example help.c:27, the window will be positioned so the indicated line is visible and selected. This feature is reminiscent of Robert Henry's error(1) program in Berkeley Unix, although it is integrated more deeply and uniformly. Also, unlike `error`, `help`'s syntax permits specifying general locations, although only line numbers will be used in this paper.

It is possible to execute any external Plan 9 command. If a command is not a built-in like `Open`, it is assumed to be an executable file and the arguments are passed to the command to be executed. For example, if one selects with the middle button the text

```
grep '^main' /sys/src/cmd/help/*.c
```

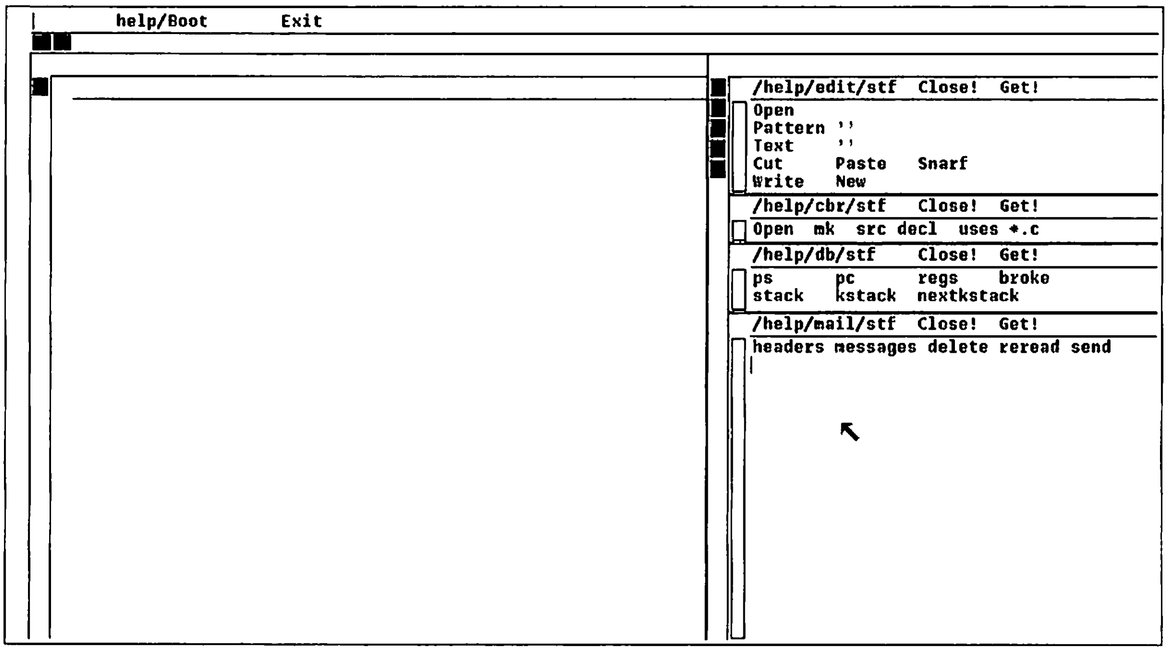
the traditional command will be executed. Again, some default rules come into play. If the tag line of the window containing the command has a file name and the command does not begin with a slash, the directory of the file will be prepended to the command. If that command cannot be found locally, it will be searched for in the standard directory of program binaries. The standard input of the commands is connected to en empty file; the standard and error outputs are directed to a special window, called `Errors`, that will be created automatically if needed. The `Errors` window is also the destination of any messages printed by the built-in commands.

The interplay and consequences of these rules are easily seen by watching the system in action.

An example

In this example I will go through the process of fixing a bug reported to me in a mail message sent by a user. Please pardon the informal first person for a while; it makes the telling easier.

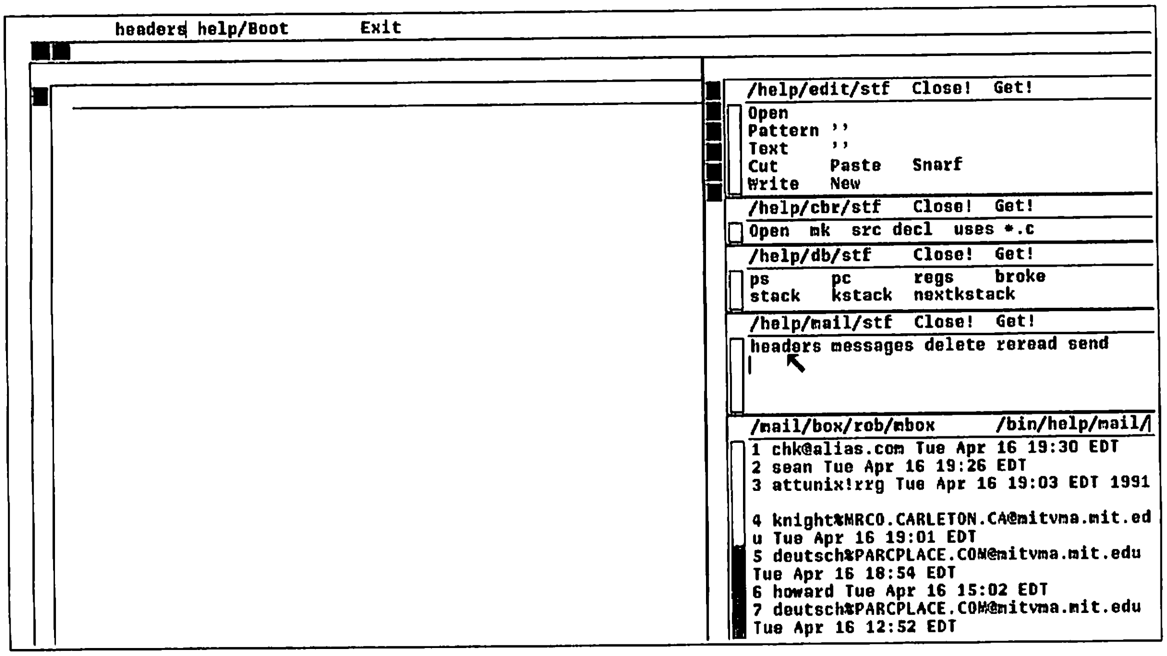
When `help` starts it loads a set of 'tools', a term borrowed from Oberon, into the right hand column of its initially two-column screen. These are files with names like `/help/edit/stf` (the stuff that the `help` editor provides), `/help/mail/stf`, and so on. Each is a plain text file that lists the names of the commands available as parts of the tool, collected in the appropriate directory. A `help` window on such a file behaves much like a menu, but is really just a window on a plain file. The useful properties stem from the interpretation of the file applied by the rules of `help`; they are not inherent to the file.



```
help/Boot        Exit

/help/edit/stf   Close!   Get!
Open
Pattern ''
Text    ''
Cut      Paste    Snarf
Write    New

/help/cbr/stf    Close!   Get!
Open  mk  src  decl  uses  *.c

/help/db/stf     Close!   Get!
ps       pc       regs     broke
stack    kstack   nextkstack

/help/mail/stf   Close!   Get!
headers messages delete reread send
```

Figure 4: The screen after booting.

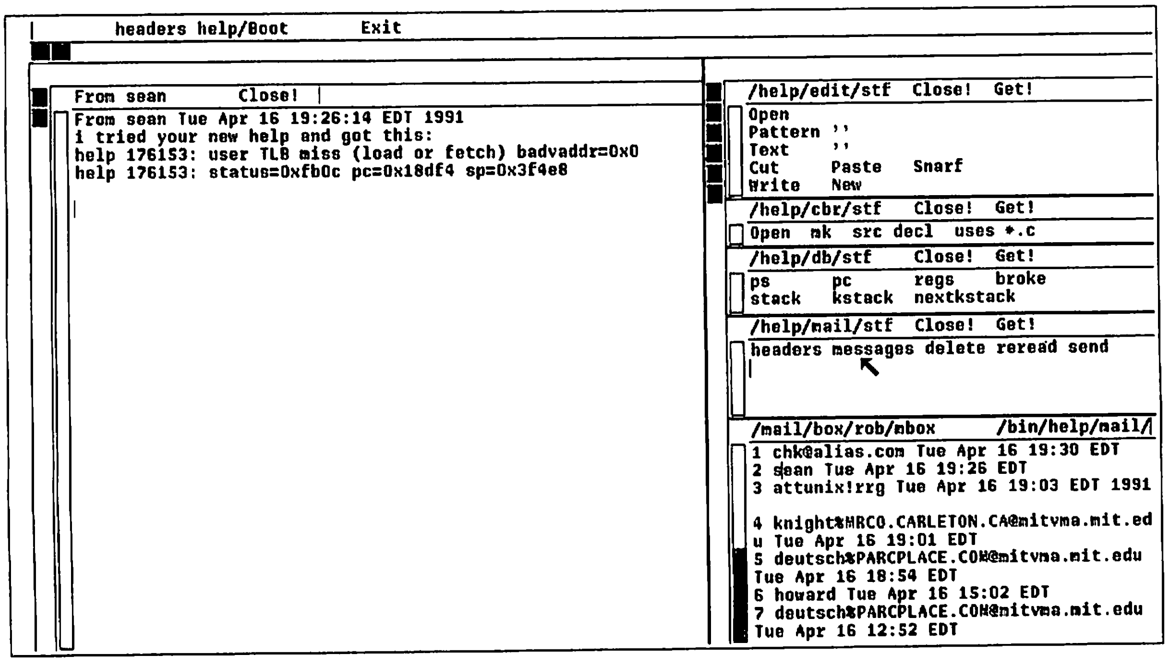
To read my mail, I first execute `headers` in the mail tool, that is, I click the middle mouse button on the word `headers` in the window containing the file `/help/mail/stf`. This executes the program `/help/mail/headers` by prefixing the directory name of the file `/help/mail/stf`, collected from the tag, to the executed word, `headers`. This simple mechanism makes it easy to manage a collection of programs in a directory.



```
headers help/Boot        Exit

/help/edit/stf   Close!   Get!
Open
Pattern ''
Text    ''
Cut       Paste    Snarf
Write     New
/help/cbr/stf    Close!   Get!
Open  mk  src decl  uses *.c
/help/db/stf     Close!   Get!
ps        pc        regs     broke
stack     kstack    nextkstack
/help/mail/stf   Close!   Get!
headers messages delete reread send

/mail/box/rob/mbox          /bin/help/mail/
1 chk@alias.com Tue Apr 16 19:30 EDT
2 sean Tue Apr 16 19:26 EDT
3 attunix!rrg Tue Apr 16 19:03 EDT 1991

4 knight%MRCO.CARLETON.CA@mitvma.mit.ed
u Tue Apr 16 19:01 EDT
5 deutsch%PARCPLACE.COM@mitvma.mit.edu
Tue Apr 16 18:54 EDT
6 howard Tue Apr 16 15:02 EDT
7 deutsch%PARCPLACE.COM@mitvma.mit.edu
Tue Apr 16 12:52 EDT
```

Figure 5: After executing mail/headers.

Headers creates a new window containing the headers of my mail messages, and labels it /mail/box/rob/mbox. I know Sean has sent me mail, so I point at the header of his mail (just pointing with the left button anywhere in the header line will do) and click on messages.

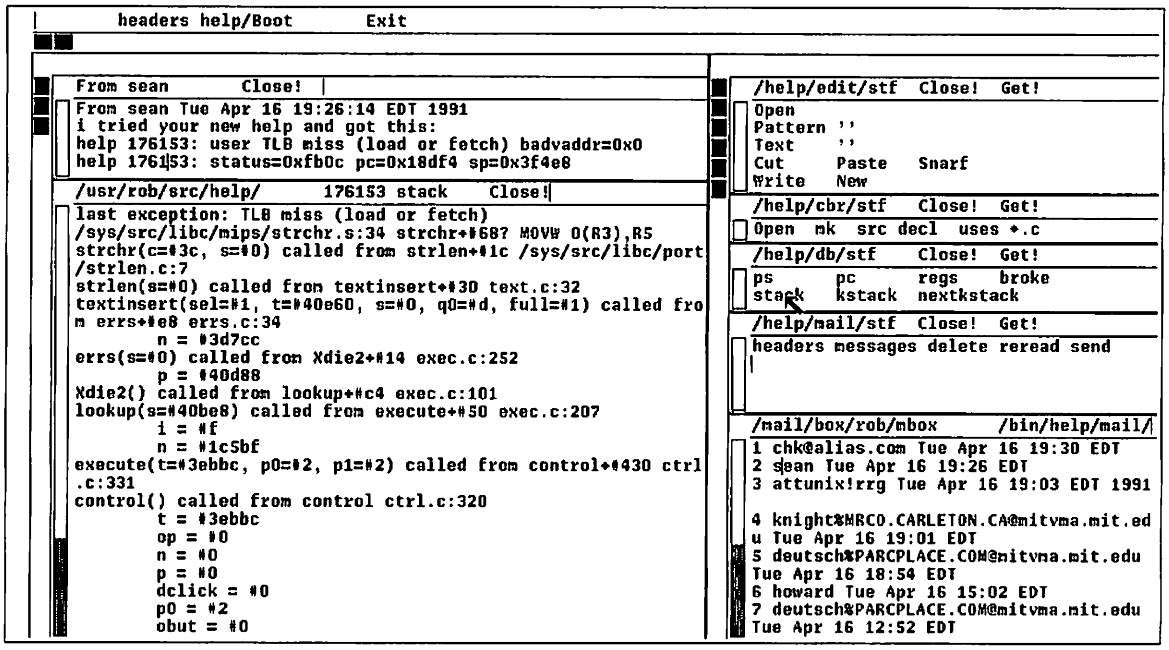


```
headers help/Boot        Exit

From sean       Close!
From sean Tue Apr 16 19:26:14 EDT 1991
i tried your new help and got this:
help 176153: user TLB miss (load or fetch) badvaddr=0x0
help 176153: status=0xfb0c pc=0x18df4 sp=0x3f4e8

/help/edit/stf   Close!   Get!
Open
Pattern ''
Text    ''
Cut       Paste    Snarf
Write     New
/help/cbr/stf    Close!   Get!
Open  mk  src decl  uses *.c
/help/db/stf     Close!   Get!
ps        pc        regs     broke
stack     kstack    nextkstack
/help/mail/stf   Close!   Get!
headers messages delete reread send

/mail/box/rob/mbox          /bin/help/mail/
1 chk@alias.com Tue Apr 16 19:30 EDT
2 sean Tue Apr 16 19:26 EDT
3 attunix!rrg Tue Apr 16 19:03 EDT 1991

4 knight%MRCO.CARLETON.CA@mitvma.mit.ed
u Tue Apr 16 19:01 EDT
5 deutsch%PARCPLACE.COM@mitvma.mit.edu
Tue Apr 16 18:54 EDT
6 howard Tue Apr 16 15:02 EDT
7 deutsch%PARCPLACE.COM@mitvma.mit.edu
Tue Apr 16 12:52 EDT
```

Figure 6: After applying messages to the header line of Sean's mail.

A new version of help has crashed and a broken process lies about waiting to be examined. (This is a property of Plan 9, not of help.) I point at the process number (I certainly shouldn't have to type it) and execute stack in the debugger tool,

/help/db/stf. This pops up a window containing the traceback as reported by adb, a primitive debugger, under the auspices of /help/db/stack.

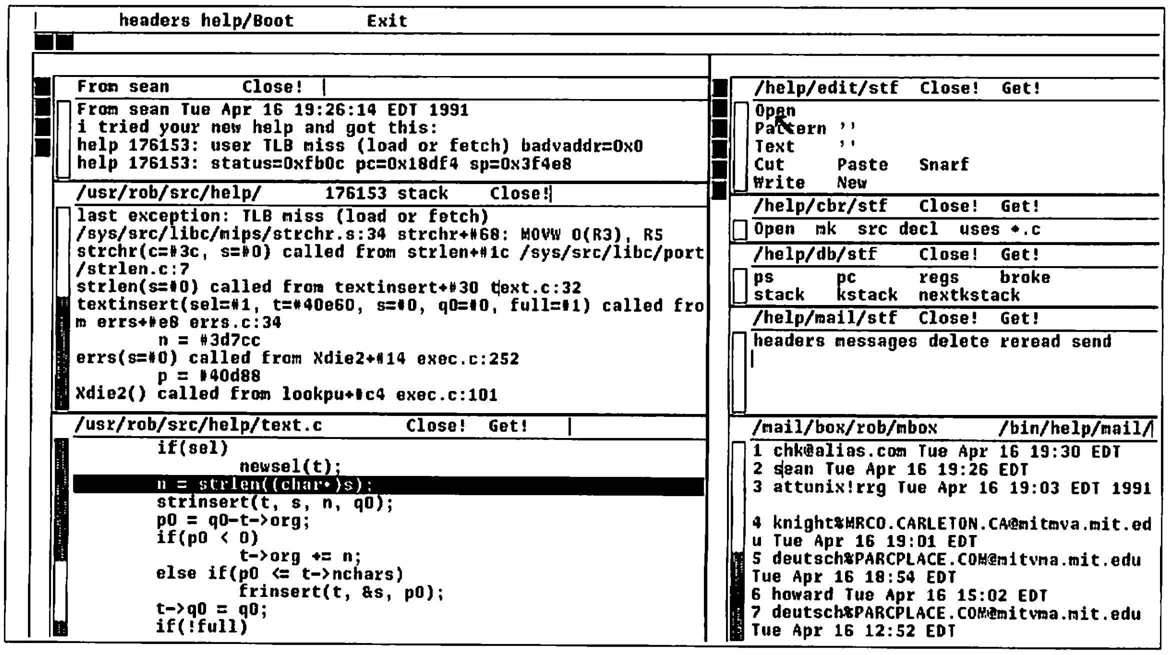


```
headers help/Boot       Exit

From sean        Close!
From sean Tue Apr 16 19:26:14 EDT 1991
i tried your new help and got this:
help 176153: user TLB miss (load or fetch) badvaddr=0x0
help 176153: status=0xfb0c pc=0x18df4 sp=0x3f4e8

/usr/rob/src/help/      176153 stack    Close!
last exception: TLB miss (load or fetch)
/sys/src/libc/mips/strchr.s:34 strchr+#68? MOVW 0(R3),R5
strchr(c=#3c, s=#0) called from strlen+#1c /sys/src/libc/port
/strlen.c:7
strlen(s=#0) called from textinsert+#30 text.c:32
textinsert(sel=#1, t=#40e60, s=#0, q0=#d, full=#1) called fro
m errs+#e8 errs.c:34
        n = #3d7cc
errs(s=#0) called from Xdie2+#14 exec.c:252
        p = #40d88
Xdie2() called from lookup+#c4 exec.c:101
lookup(s=#40be8) called from execute+#50 exec.c:207
        i = #f
        n = #1c5bf
execute(t=#3ebbc, p0=#2, p1=#2) called from control+#430 ctrl
.c:331
control() called from control ctrl.c:320
        t = #3ebbc
        op = #0
        n = #0
        p = #0
        dclick = #0
        p0 = #2
        obut = #0
```

```
/help/edit/stf  Close!  Get!
Open
Pattern ''
Text    ''
Cut     Paste   Snarf
Write   New

/help/cbr/stf   Close!  Get!
Open  mk  src decl  uses *.c

/help/db/stf    Close!  Get!
ps      pc      regs    broke
stack   kstack  nextkstack

/help/mail/stf  Close!  Get!
headers messages delete reread send

/mail/box/rob/mbox      /bin/help/mail/
1 chk@alias.com Tue Apr 16 19:30 EDT
2 sean Tue Apr 16 19:26 EDT
3 attunix!rrg Tue Apr 16 19:03 EDT 1991

4 knight%MRCO.CARLETON.CA@mitvma.mit.ed
u Tue Apr 16 19:01 EDT
5 deutsch%PARCPLACE.COM@mitvma.mit.edu
Tue Apr 16 18:54 EDT
6 howard Tue Apr 16 15:02 EDT
7 deutsch%PARCPLACE.COM@mitvma.mit.edu
Tue Apr 16 12:52 EDT
```

Figure 7: After applying db/stack to the broken process.

Notice that this new window has many file names in it. These are extracted from the symbol table of the broken program. I can look at the line (of assembly language) that died by pointing at the entry /sys/src/libc/mips/strchr.s:34 and executing Open, but I'm sure the problem lies further up the call stack. The deepest routine in help is textinsert, which calls strlen on line 32 of the file text.c. I point at the identifying text in the stack window and execute Open to see the source.

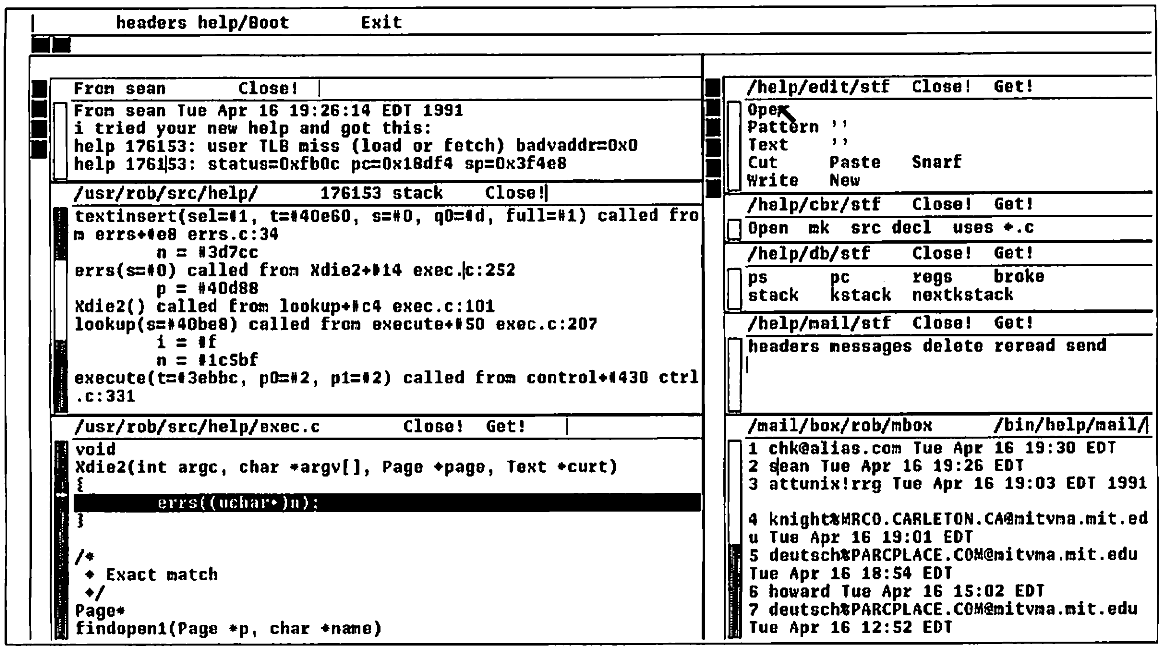


```
headers help/Boot       Exit

From sean        Close!
From sean Tue Apr 16 19:26:14 EDT 1991
i tried your new help and got this:
help 176153: user TLB miss (load or fetch) badvaddr=0x0
help 176153: status=0xfb0c pc=0x18df4 sp=0x3f4e8

/usr/rob/src/help/      176153 stack    Close!
last exception: TLB miss (load or fetch)
/sys/src/libc/mips/strchr.s:34 strchr+#68: MOVW 0(R3), R5
strchr(c=#3c, s=#0) called from strlen+#1c /sys/src/libc/port
/strlen.c:7
strlen(s=#0) called from textinsert+#30 text.c:32
textinsert(sel=#1, t=#40e60, s=#0, q0=#0, full=#1) called fro
m errs+#e8 errs.c:34
        n = #3d7cc
errs(s=#0) called from Xdie2+#14 exec.c:252
        p = #40d88
Xdie2() called from lookpu+#c4 exec.c:101

/usr/rob/src/help/text.c        Close!  Get!
        if(sel)
                newsel(t);
        n = strlen((char*)s);
        strinsert(t, s, n, q0);
        p0 = q0-t->org;
        if(p0 < 0)
                t->org += n;
        else if(p0 <= t->nchars)
                frinsert(t, &s, p0);
        t->q0 = q0;
        if(!full)
```

```
/help/edit/stf  Close!  Get!
Open
Pattern ''
Text    ''
Cut     Paste   Snarf
Write   New

/help/cbr/stf   Close!  Get!
Open  mk  src decl  uses *.c

/help/db/stf    Close!  Get!
ps      pc      regs    broke
stack   kstack  nextkstack

/help/mail/stf  Close!  Get!
headers messages delete reread send

/mail/box/rob/mbox      /bin/help/mail/
1 chk@alias.com Tue Apr 16 19:30 EDT
2 sean Tue Apr 16 19:26 EDT
3 attunix!rrg Tue Apr 16 19:03 EDT 1991

4 knight%MRCO.CARLETON.CA@mitmva.mit.ed
u Tue Apr 16 19:01 EDT
5 deutsch%PARCPLACE.COM@mitvma.mit.edu
Tue Apr 16 18:54 EDT
6 howard Tue Apr 16 15:02 EDT
7 deutsch%PARCPLACE.COM@mitvma.mit.edu
Tue Apr 16 12:52 EDT
```

Figure 8: After Opening text.c at line 32.

The problem is coming to light: s, the argument to strlen, is zero, and was passed as an argument to textinsert by the routine errs, which apparently also got it as an argument from Xdie2. I close the window on text.c by hitting Close! in the tag of the window. By convention, commands ending in an exclamation mark take no arguments; they are window operations that apply to the window in which they are executed. Next I examine the source of the suspiciously named Xdie2 by pointing at the stack trace and Opening again. (See Figure 9.)



```
headers help/Boot        Exit

From sean        Close!
From sean Tue Apr 16 19:26:14 EDT 1991
i tried your new help and got this:
help 176153: user TLB miss (load or fetch) badvaddr=0x0
help 176153: status=0xfb0c pc=0x18df4 sp=0x3f4e8

/usr/rob/src/help/      176153 stack     Close!
textinsert(sel=#1, t=#40e60, s=#0, q0=#d, full=#1) called fro
m errs+#e8 errs.c:34
        n = #3d7cc
errs(s=#0) called from Xdie2+#14 exec.c:252
        p = #40d88
Xdie2() called from lookup+#c4 exec.c:101
lookup(s=#40be8) called from execute+#50 exec.c:207
        i = #f
        n = #1c5bf
execute(t=#3ebbc, p0=#2, p1=#2) called from control+#430 ctrl
.c:331

/usr/rob/src/help/exec.c          Close!  Get!
void
Xdie2(int argc, char *argv[], Page *page, Text *curt)
{
        errs((uchar*)n);
}

/*
 * Exact match
 */
Page*
findopen1(Page *p, char *name)
```

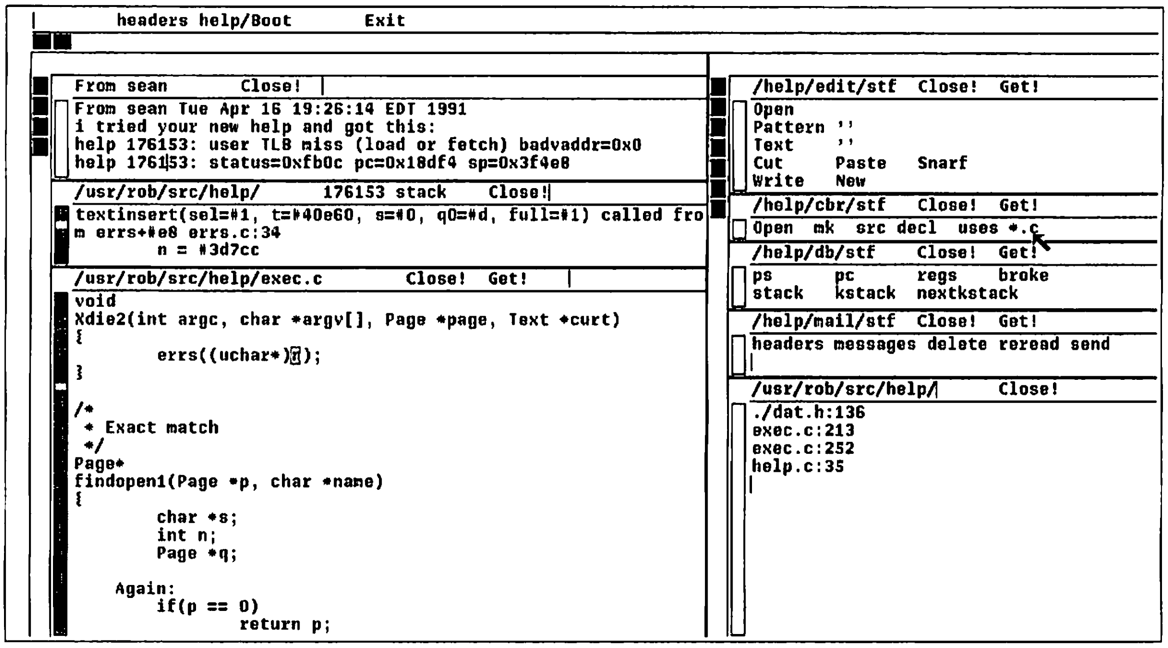
```
/help/edit/stf  Close!  Get!
Open
Pattern ''
Text    ''
Cut     Paste   Snarf
Write   New

/help/cbr/stf   Close!  Get!
Open  mk  src decl  uses *.c

/help/db/stf    Close!  Get!
ps      pc      regs    broke
stack   kstack  nextkstack

/help/mail/stf  Close!  Get!
headers messages delete reread send

/mail/box/rob/mbox      /bin/help/mail/
1 chk@alias.com Tue Apr 16 19:30 EDT
2 sean Tue Apr 16 19:26 EDT
3 attunix!rrg Tue Apr 16 19:03 EDT 1991

4 knight%MRCO.CARLETON.CA@mitvma.mit.ed
u Tue Apr 16 19:01 EDT
5 deutsch%PARCPLACE.COM@mitvma.mit.edu
Tue Apr 16 18:54 EDT
6 howard Tue Apr 16 15:02 EDT
7 deutsch%PARCPLACE.COM@mitvma.mit.edu
Tue Apr 16 12:52 EDT
```

Figure 9: After Opening exec.c at line 252.

Now the problem gets harder. The argument passed to errs is a variable, n, that appears to be global. Who set it to zero? I can look at all the uses of the variable in the program by pointing at the variable in the source text and executing uses *.c by sweeping both 'words' with the middle button in the C browser tool, /help/cbr/stf. Uses creates a new window with all references to the variable n in the files /usr/rob/src/help/*.c indicated by file name and line number. The implementation of the C browser is described below; in a nutshell, it parses the C source to interpret the symbols dynamically. If instead I had run the regular Unix command

```
grep n /usr/rob/src/help/*.c
```

I would have had to wade through every occurrence of the letter n in the program.



```
headers help/Boot        Exit
```

```
From sean        Close!
From sean Tue Apr 16 19:26:14 EDT 1991
i tried your new help and got this:
help 176153: user TLB miss (load or fetch) badvaddr=0x0
help 176153: status=0xfb0c pc=0x18df4 sp=0x3f4e8
```

```
/usr/rob/src/help/       176153 stack       Close!
textinsert(sel=#1, t=#40e60, s=#0, q0=#d, full=#1) called fro
m errs+#e8 errs.c:34
        n = #3d7cc
```

```
/usr/rob/src/help/exec.c          Close!  Get!
void
Xdie2(int argc, char *argv[], Page *page, Text *curt)
{
        errs((uchar*)▯);
}

/*
 * Exact match
 */
Page*
findopen1(Page *p, char *name)
{
        char *s;
        int n;
        Page *q;

    Again:
        if(p == 0)
                return p;
```

```
/help/edit/stf   Close!  Get!
Open
Pattern ''
Text    ''
Cut     Paste    Snarf
Write   New
```

```
/help/cbr/stf    Close!  Get!
Open  mk  src decl  uses *.c
```

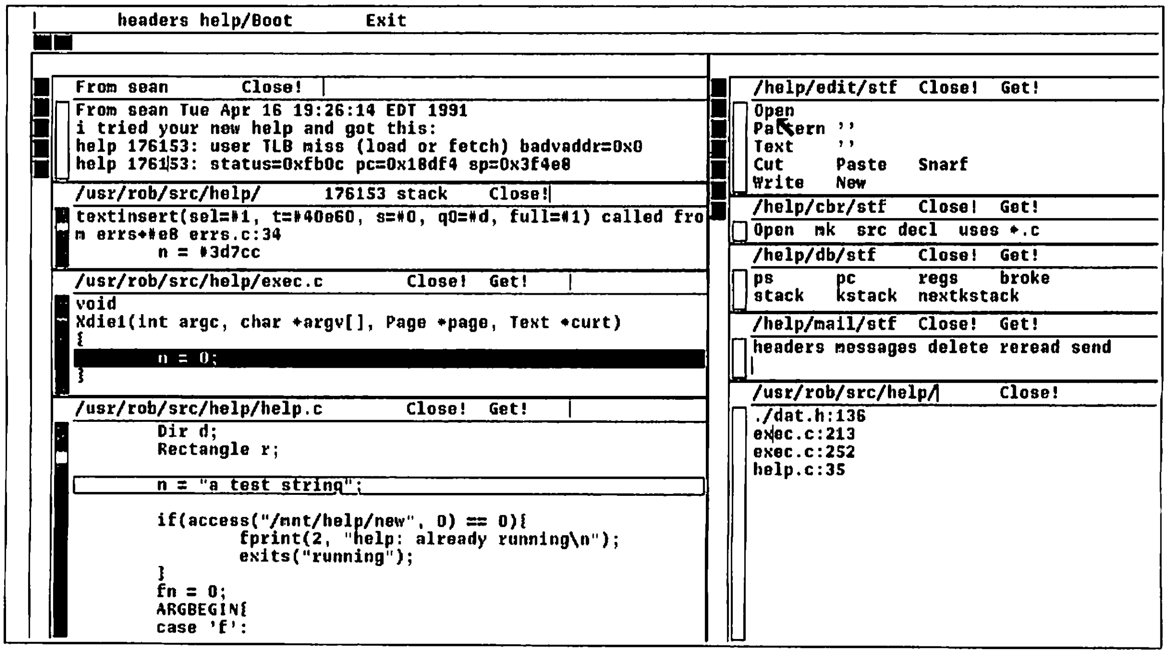
```
/help/db/stf     Close!  Get!
ps      pc      regs    broke
stack   kstack  nextkstack
```

```
/help/mail/stf   Close!  Get!
headers messages delete reread send
```

```
/usr/rob/src/help/       Close!
./dat.h:136
exec.c:213
exec.c:252
help.c:35
```

Figure 10: After finding all uses of n.

The first use is clearly the declaration in the header file. It looks like `help.c:35` should be an initialization. I `Open` `help.c` to that line and see that the variable is indeed initialized. (See Figure 11; a few lines off the top of the window on `help.c` is the opening declaration of `main()`.) Some other use of n must have cleared it. Line 252 of `exec.c` is the call; I know that's a read, not a write, of the variable. So I point to `exec.c:213` and execute `Open`.



```
headers help/Boot        Exit
```

```
From sean        Close!
From sean Tue Apr 16 19:26:14 EDT 1991
i tried your new help and got this:
help 176153: user TLB miss (load or fetch) badvaddr=0x0
help 176153: status=0xfb0c pc=0x18df4 sp=0x3f4e8
```

```
/usr/rob/src/help/       176153 stack       Close!
textinsert(sel=#1, t=#40e60, s=#0, q0=#d, full=#1) called fro
m errs+#e8 errs.c:34
        n = #3d7cc
```

```
/usr/rob/src/help/exec.c          Close!  Get!
void
Xdie1(int argc, char *argv[], Page *page, Text *curt)
{
        n = 0;
}
```

```
/usr/rob/src/help/help.c          Close!  Get!
        Dir d;
        Rectangle r;

        n = "a test string";

        if(access("/mnt/help/new", 0) == 0){
                fprint(2, "help: already running\n");
                exits("running");
        }
        fn = 0;
        ARGBEGIN{
        case 'f':
```

```
/help/edit/stf   Close!  Get!
Open
Pattern ''
Text    ''
Cut     Paste    Snarf
Write   New
```

```
/help/cbr/stf    Close!  Get!
Open  mk  src decl  uses *.c
```

```
/help/db/stf     Close!  Get!
ps      pc      regs    broke
stack   kstack  nextkstack
```

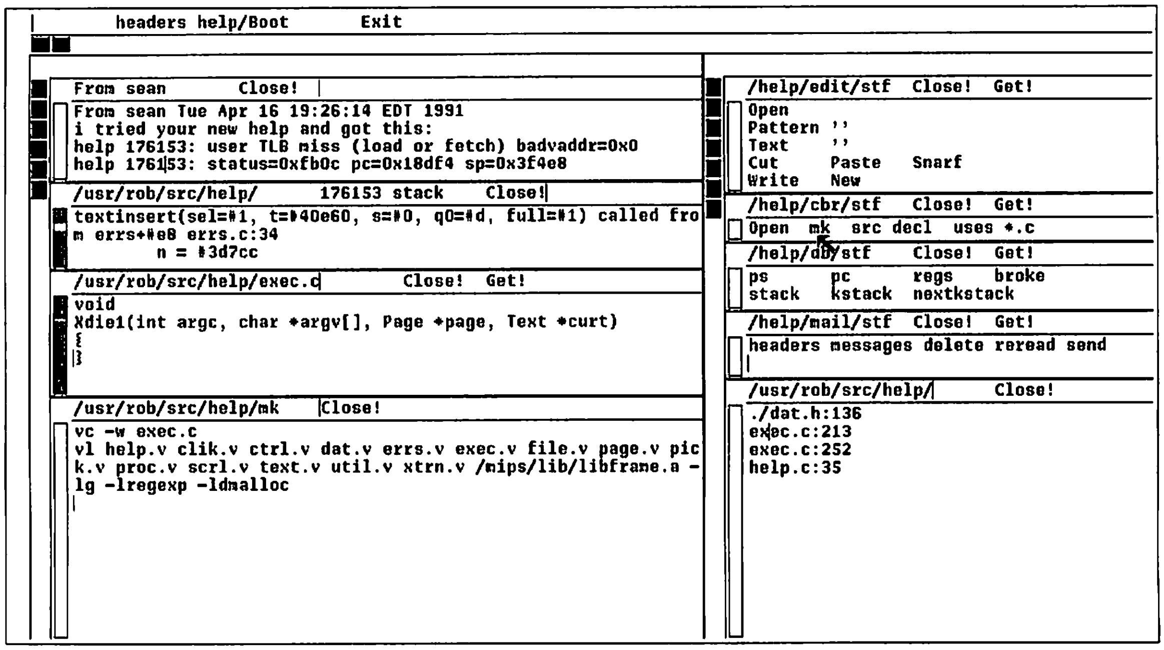
```
/help/mail/stf   Close!  Get!
headers messages delete reread send
```

```
/usr/rob/src/help/       Close!
./dat.h:136
exec.c:213
exec.c:252
help.c:35
```

Figure 11: The writing of n on line `exec.c:213`.

Here is the jackpot of this contrived example. Sometime before `Xdie2` was executed, `Xdie1` cleared n. I use `Cut` to remove the offending line, write the file back out (the word `Put!` appears in the tag of a modified window) and then execute `mk` in

/help/cbr to compile the program (a total of three clicks of the middle button). I could now answer Sean's mail to tell him that the bug is fixed. I'll stop now, though, because to answer his mail I'd have to type something. Through this entire demo I haven't yet touched the keyboard.



```
headers help/Boot        Exit

From sean        Close!
From sean Tue Apr 16 19:26:14 EDT 1991
i tried your new help and got this:
help 176153: user TLB miss (load or fetch) badvaddr=0x0
help 176153: status=0xfb0c pc=0x18df4 sp=0x3f4e8

/usr/rob/src/help/     176153 stack     Close!
textinsert(sel=#1, t=#40e60, s=#0, q0=#d, full=#1) called fro
m errs+#e8 errs.c:34
        n = #3d7cc

/usr/rob/src/help/exec.c          Close!  Get!
void
Xdie1(int argc, char *argv[], Page *page, Text *curt)
{
}

/usr/rob/src/help/mk     Close!
vc -w exec.c
vl help.v clik.v ctrl.v dat.v errs.v exec.v file.v page.v pic
k.v proc.v scrl.v text.v util.v xtrn.v /mips/lib/libframe.a -
lg -lregexp -ldmalloc
```

```
/help/edit/stf  Close!  Get!
Open
Pattern ''
Text    ''
Cut     Paste   Snarf
Write   New

/help/cbr/stf   Close!  Get!
Open  mk  src decl  uses *.c

/help/db/stf    Close!  Get!
ps      pc      regs    broke
stack   kstack  nextkstack

/help/mail/stf  Close!  Get!
headers messages delete reread send

/usr/rob/src/help/      Close!
./dat.h:136
exec.c:213
exec.c:252
help.c:35
```

Figure 12: After the program is compiled.

This demonstration illustrates several things besides the general flavor of help. Most important, by following some simple rules it is possible to build an extremely efficient and productive user interface using just a mouse and screen. This is illustrated by how help makes it easy to work with files and commands in multiple directories. The rules by which help constructs file names from context and by which the utilities derive the context in which they execute simplify the management of programs and other systems constructed from scattered components. Also, the few common rules about text and file names allow a variety of applications to interact through a single user interface. For example, none of the tool programs has any code to interact directly with the keyboard or mouse. Instead help passes to an application the file and character offset of the mouse position. Using the interface described in the next section, the application can then examine the text in the window to see what the user is pointing at. These operations are easily encapsulated in simple shell scripts, an example of which is given below.

The interface seen by programs

As in 8½, the Plan 9 window system [Pike91], help provides its client processes access to its structure by presenting a file service, although help's file structure is very different. Each help window is represented by a set of files stored in numbered directories. The number is a unique identifier, similar to Unix process id's. Each directory contains files such as tag and body, which may be read to recover the contents of the corresponding subwindow, and ctl, to which may be written messages to effect changes such as insertion and deletion of text in contents of the window. The help directory is conventionally mounted at /mnt/help, so to copy the text in the body of window number 7 to a file, one may execute

```
cp /mnt/help/7/body file
```

To search for a text pattern,

```
grep pattern /mnt/help/7/body
```

An ASCII file /mnt/help/index may be examined to connect tag file names to window numbers. Each line of this file is a window number, a tab, and the first line of the tag.

To create a new window, a process just opens /mnt/help/new/ctl, which places the new window automatically on the screen near the current selected text, and may then read from that file the name of the window created, e.g. /mnt/help/8. The position and size of the new window is chosen by help.

Another example

The directory /help/cbr contains the C browser we used above. One of the programs there is called decl; it finds the declaration of the variable marked by the selected text. Thus one points at a variable with the left button and then executes decl in the window for the file /help/cbr/stf. Help executes /help/cbr/decl using the context rules for the *executed* text and passes it the context (window number and location) of the *selected* text through an environment variable, helpsel.

Decl is a shell script, a program for the Plan 9 shell, rc [Duff90]. Here is the complete script:

```
eval `{help/parse -c}
x=`{cat /mnt/help/new/ctl}
{
    echo a
    echo $dir/'    Close!'
} | help/buf > /mnt/help/$x/ctl
{
    cpp $cppflags $file |
        help/rcc -w -g -i$id -n$line |
        sed 1q
} > /mnt/help/$x/bodyapp
```

The first line runs a small program, help/parse, that examines $helpsel and establishes another set of environment variables, file, id, and line, describing what the user is pointing at. The next creates a new window and sets x to its number. The first block writes the directory name to the tag line; the second runs the C preprocessor on the original source file (it should arguably be run on, say, /mnt/help/8/body) and passes the resulting text to a special version of the compiler. This compiler has no code generator; it parses the program and manages the symbol table, and when it sees the declaration for the indicated identifier on the appropriate line of the file, it prints the file coordinates of that declaration. This appears on standard output, which is appended to the new window by writing to /mnt/help/$x/bodyapp. The user can then point at the output to direct Open to display the appropriate line in the source. (A future change to help will be to close this loop so the Open operation also happens automatically.) Thus with only three button clicks one may fetch to the screen the declaration, from whatever file in which it resides, the declaration of a variable, function, type, or any other C object.

A couple of observations about this example. First, help provided all the user interface. To turn a compiler into a browser involved spending a few hours stripping the code generator from the compiler and then writing a half dozen brief shell scripts to connect it up to the user interface for different browsing functions. Given another language, we would need only to modify the compiler to achieve the same result. *We would not need to write any user interface software.* Second, the resulting application is not a monolith. It is instead a small suite of tiny shell scripts that may be tuned or toyed with for other purposes or experiments.

Other applications are similarly designed. For example, the debugger interface, /help/db, is a directory of ten or so brief shell scripts, about a dozen lines each, that connect adb to help. Adb has a notoriously cryptic input language; the commands in /help/db package the most important functions of adb as easy-to-use operations that connect to the rest of the system while hiding the rebarbative syntax. People unfamiliar with adb can easily use help's interface to it to examine broken processes. Of course, this is hardly a full-featured debugger, but it was written in about an hour and illustrates the principle. It is a prototype, and help is an easy-to-program environment in which to build such test programs. A more sophisticated debugger could be assembled in a similar way, perhaps by leaving a debugging process resident in the background and having the help commands send it requests.

Discussion

Help is a research prototype that explores some ideas in user interface design. As an experiment it has been successful. When someone first begins to use help, the profusion of windows and the different ground rules for the user interface are disorienting. After a couple of hours, though, the system seems seductive, even natural. To return at that point to a more traditional environment is to see how much smoother help really is. Unfortunately, it is sometimes necessary to leave help because of its limitations.

The time is overdue to rewrite help with an eye to such mundane but important features as undo, multiple windows per file, the ability to handle large files gracefully, support for traditional shell windows, and syntax for shell-like functionality such as I/O redirection. Also, of course, the restriction to textual applications should be eliminated.

One of the original problems with the system — inadequate heuristics for automatically placing windows — has been fixed since the first version of this paper. The rule it follows is first to place the new window at the bottom of the column containing the selection. It places the tag of the window immediately below the lowest visible text already in the column. If that would leave too little of the new window visible, the new window is placed to cover half of the lowest window in the column. If that would still leave too little visible, the new window is positioned over the bottom 25% of the column and minor adjustments are made so it covers no partial line of existing text, which may entail hiding some windows entirely. This procedure is good enough that I haven't been encouraged to refine it any further, although there are probably improvements that could still be made. A good rule to follow when designing or tuning interfaces is to attend to any clumsiness that draws your attention to the interface and distracts from the job at hand. I believe the heuristic for placing windows is good enough because I don't notice it; in fact I had to read the source to help to recall what it was.

Help does not exploit the multi-machine Plan 9 environment as well as it could. The most obvious example is that the applications run on the same machine as help itself. This is probably easy to fix: help could run on the terminal and make an invisible call to the CPU server, sending requests to run applications to the remote shell-like process. This is similar to how nmake [Fowl90] runs its subprocesses.

If imitation is the sincerest form of flattery, the designers of Oberon's user interface will (I hope) be honored by help. But Oberon has some aspects that made it difficult to adapt the user interface directly to UNIX-like systems such as Plan 9. The most important is that Oberon is a monolithic system intimately tied to a module-based language. An

Oberon tool, for instance, is essentially just a listing of the entry points of a module. In retrospect, the mapping of this idea into commands in a Unix directory may seem obvious, but it took a while to discover. Once it was found, the idea to use the directory name associated with a file or window as a context, analogous to the Oberon module, was a real jumping-off point. `Help` only begins to explore its ramifications.

Another of Oberon's difficulties is that it is a single-process system. When an application is running, all other activity — even mouse tracking — stops. It turned out to be easy to adapt the user interface to a multi-process system. `Help` may even be superior in this regard to traditional shells and window systems since it makes a clean separation between the text that executes a command and the result of this command. When windows are cheap and easy to use why not just create a window for every process? Also, `help`'s structure as a Plan 9 file server makes the implementation of this sort of multiplexing straightforward.

`Help` is similar to a hypertext system, but the connections between the components are not in the data — the contents of the windows — but rather in the way the system itself interprets the data. When information is added to a hypertext system, it must be linked (often manually) to the existing data to be useful. Instead, in `help`, the links form automatically and are context-dependent. In a session with `help`, things start slowly because the system has little text to work with. As each new window is created, however, it is filled with text that points to new and old text, and a kind of exponential connectivity results. After a few minutes the screen is filled with active data. Compare Figure 4 to Figure 11 to see snapshots of this process in action. `Help` is more dynamic than any hypertext system for software development that I have seen. (It is also smaller: 4300 lines of C.)

The main area where `help` has not been pushed hard enough is, in fact, its intended subject: software development. The focus has been more on the user interface than on how it is used. One of the applications that should be explored is compilation control. Running `make` in the appropriate directory is too pedestrian for an environment like this. Also, for complicated trees of source directories, the `makefiles` would need to be modified so the file names would couple well with `help`'s way of working. `Make` and `help` don't function in similar ways. `Make` works by being told what target to build and looking at which files have been changed that are components of the target. What's needed for `help` is almost the opposite: a tool that, perhaps by examining the `index` file, sees what source files have been modified and builds the targets that depend on them. Such a program may be a simple variation of `make` — the information in the `makefile` would be the same — or it may be a whole new tool. Either way, it should be possible to tighten the binding between the compilation process and the editing of the source code; deciding what work to do by noticing file modification times is inelegant.

There have been other recent attempts to integrate a user interface more closely with the applications and the operating system. ConMan and Tcl [Haeb88,Oust90] are noteworthy examples, but they just provide interprocess communication within existing environments, permitting established programs to talk to one another. `Help` is more radical. It provides the entire interface to the screen and mouse for both users and programs. It is not an extra layer of software above the window system; instead it replaces the window system, the toolkits, the command interpreter, the editor, and even the user interface code within the applications.

Perhaps its most radical idea, though, is that a better user interface can be one with fewer features. `Help` doesn't even have pop-up menus; it makes them superfluous. It has no decorations, no pictures, and no modes, yet by using only a bitmap screen and three mouse buttons (one of which is underused) it provides a delightfully snappy and natural user interface, one that makes regular window systems — including those I have written — seem heavy-handed. `Help` demonstrates that the ideas of minimalism, uniformity, and universality have merit in the design of human-computer interfaces. In the years to come, as the machines and their input methods become more complex, those principles will have to be followed ever more assiduously if we are to get the most from our systems.

Acknowledgements

Sean Dorward wrote the mail tools and suggested many improvements to `help`. Doug Blewett, Tom Duff, Stu Feldman, Eric Grosse, Dennis Ritchie, and Howard Trickey made helpful comments on the paper. Brian Kernighan's heroic efforts to force this paper through `troff` deserve particular thanks.

References

[Duff90] Tom Duff, "Rc - A Shell for Plan 9 and UNIX systems", Proc. of the Summer 1990 UKUUG Conf., London, July, 1990, pp. 21-33

[Fowl90] Glenn Fowler, "A Case for make", Softw. - Prac. and Exp., Vol 20 #S1, June 1990, pp. 30-46

[Haeb90] Paul Haeberli, "ConMan: A Visual Programming Language for Interactive Graphics", Comp. Graph., Vol 22 #4, Aug. 1988, pp. 103-110

[Oust90] John Ousterhout, "Tcl: An Embeddable Command Language", Proc. USENIX Winter 1990 Conf., pp. 133-146

[Pike88] Rob Pike, "Window Systems Should Be Transparent", Comp. Sys., Summer 1988, Vol 1 #3, pp. 279-296

[Pike90] Rob Pike, Dave Presotto, Ken Thompson, and Howard Trickey, "Plan 9 from Bell Labs", Proc. of the Summer 1990 UKUUG Conf., London, July, 1990, pp. 1-9

[Pike91] Rob Pike, "8½, the Plan 9 Window System", USENIX Summer Conf. Proc., Nashville, June, 1991, pp. 257-265

[Reis91] Martin Reiser, *The Oberon System*, Addison Wesley, New York, 1991

[Wirt89] N. Wirth and J. Gutknecht, "The Oberon System", Softw. - Prac. and Exp., Sep 1989, Vol 19 #9, pp 857-894

A Multi-Layer Graphic Model for Building Interactive Graphical Applications

Jean-Daniel Fekete

LRI - CNRS URA410
Bat 490
Université de Paris-Sud
F-91405 ORSAY Cedex
+33 (1) 69 41 65 91

2001 S.A.
45, rue Camille Desmoulins
F-94230 CACHAN
+33 (1) 45 46 10 00
jdf@lri.lri.fr

Abstract

In this article, we present a model for building interactive graphical applications based on multi-layer graphics. With this model, a variety of components of an interactive graphical editor (transient objects like the selection rectangle, selected objects, grids, cursors), as well as the handling of input events, are realized in a more straightforward way.
We show how an interactive graphical application can use layered graphics in a simple and effective way to represent direct manipulation, some graphic constraints and event handling using various input devices. By clarifying the status of abstract elements of an interactive application, the model encompasses a significant part of the dynamic aspect of the interaction.

Résumé

Dans cet article, nous présentons un modèle basé sur le graphique multi-couches pour le développement d'applications graphiques interactives. En utilisant ce modèle, il est possible d'attribuer un statut clair à de nombreux éléments composant un éditeur graphique interactif (les objets «fugaces» comme le rectangle utilisé pour sélectionner, les objets sélectionnés, les grilles d'alignement ainsi que les curseurs).
Nous montrons comment une application graphique interactive peut tirer profit du modèle graphique multi-couches pour décrire simplement la manipulation directe, certaines contraintes graphiques ainsi que la gestion des événements produits par divers périphériques d'entrée. En attribuant un statut clair à ces composantes abstraites des applications graphiques interactives, le modèle aide à la construction d'une partie importante de la description du comportement dynamique de l'interaction.

KEYWORDS: User Interface Design, Interface Metaphors, Input Devices.

Introduction

Building interactive graphical programs is still a difficult task. Some toolkits [14, 11] propose a framework for building task-specific editors, but the kind of editors they can produce are stereotypical in several aspects:

- they only consider keyboard and mouse input,
- their graphic model is tied to the PostScript model,
- their interaction is based on stereotypical graphic objects like button boxes and menus,
- they offer only stereotypical mouse- and keyboard-based manipulations.

We believe that these limitations are currently too strong because of new, easily available input devices; both 2D (Wacom pressure sensitive styli, touch screens with multi touch sensitivity, eye trackers) and 3D (data gloves, flying mice, 3D styli). Since the list of devices recognized by current toolkits is "hardwired" into the toolkit, almost no support is provided for application programmers to handle these new kinds of devices for input and to support graphic feedback for their actions.
We propose here a model based on multi-layered graphics and multi-layered input handling.
This model is also suitable for describing traditional mouse- and keyboard-based interactive programs, simplifying the handling of interaction, the handling of transient shapes, of selected graphic objects, of constraint representations (*e.g.* a grid) and of device feedback.
Most graphical toolkits distinguish two levels for displaying graphics: a virtual surface where the graphics are drawn and one or several viewports which show some part of the virtual surface (see fig. 1).
In this model, the graphics on the virtual surface are produced by a **graphic controller**, a module which transforms an abstract data structure into a graphic data structure which can be displayed. Passing from the graphic data structure to the virtual surface is done by a **renderer** module (see fig. 2). Projecting the virtual surface onto a visible viewport is done through a transformation and a clipping zone. This model is used by most graphical systems and the mechanisms to

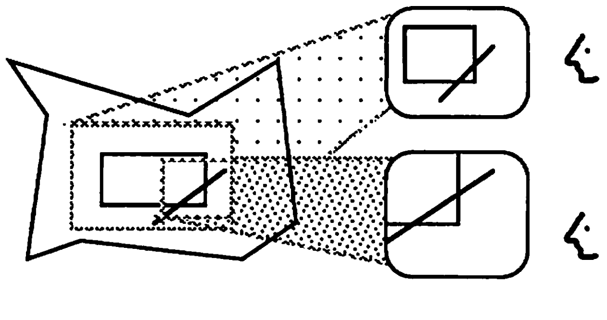


figure 1: the two levels of graphics in toolkits

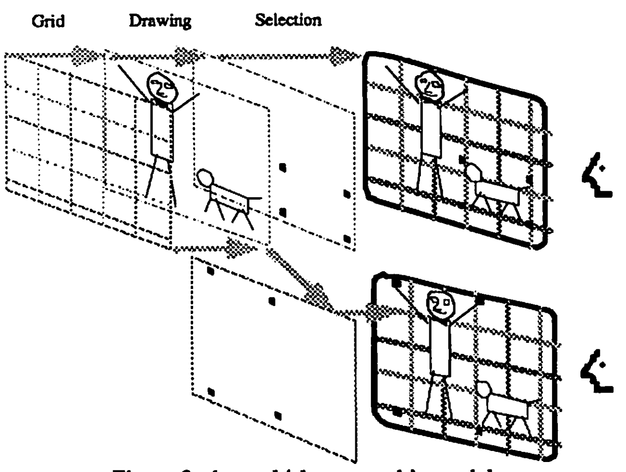


Figure 3: the multi-layer graphic model.

implement it can be found in windowing systems like X [12], the Macintosh Toolbox [2], and by graphical toolkits like InterViews [9], ET++ [15] or Garnet [11].
This model is well adapted to the display of graphics but is of little help for interaction. For instance, transient objects, like handles or sliding shapes, have no clear graphic status. They do not belong to the graphic data structure so their display and management is done by an internal part of the toolkit. This internal part is hard to adapt to new interaction and appearance styles.

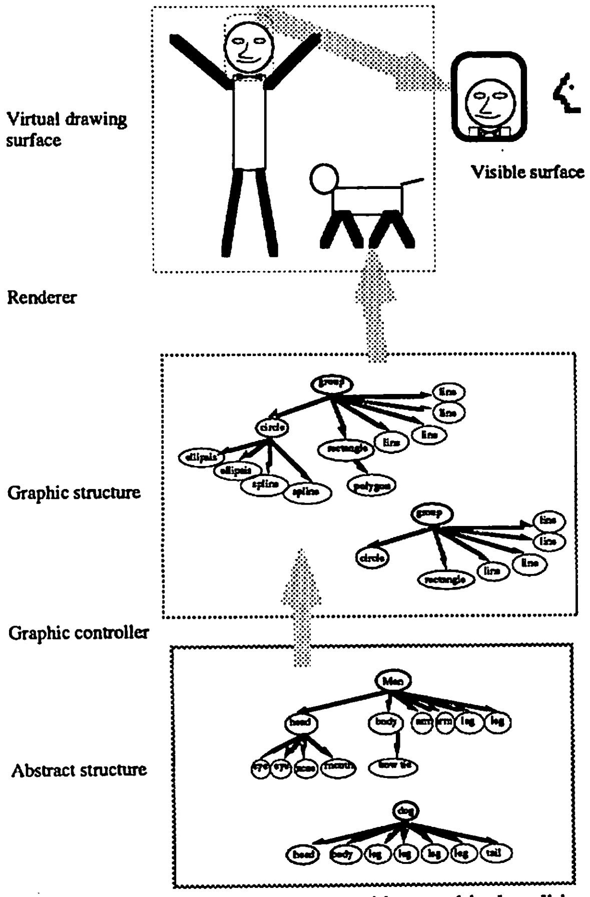


Figure 2: Transformation stages used by graphical toolkits

The Multi-Layered Graphic Model

We propose a model where the visible surface can display several virtual surfaces, like stacked transparent celluloids (see figure 3); a virtual surface supports a graphical structure. The same virtual surface can be shown on several different visible surfaces, at different levels. This is an extension to the multi-view concept.
This model is useful to specify both the display of graphics and the handling of events. Layers are drawn from the bottom (background) to the top (foreground) whereas events are handled from the topmost layer to the bottommost. In the following sections, we discuss the use of layers in typical applications, the handling of graphics, event handling, the status of various components in our model and finally an example of its use.

Layers

While using the Xtv toolkit developed at LRI [3], experiments with several interactive graphical layouts led us to distinguish between the following layers:

- *Background/model layer*: this layer displays a background image. Conceptually, areas covered by no other layer contain the background image. Window systems usually handle this layer in a special way. X, for instance, has a background image (generally a solid color) for each window. In programs like Illustrator™, this layer can contain a drawing which is then used as a template for the main drawing, like with tracing paper. In our model, this layer shows "default" graphics and handles events ignored by the other layers.
- *Graphical constraints visualization layer*: this layer usually displays a grid or other graphical formalism for representing geometrical constraints.
- *Application data layer*: this layer, as in figure 1, displays the graphical objects representing the application's internal objects. Event handling in this layer is described later.
- *Selected objects layer*: this layer displays the selection, using shapes (*e.g.* handles) expressing the kinds of manipulations available. Event handling in this layer supports direct manipulation.
- *Lexical operations representation layer*: this layer displays shapes expressing the status of input devices, like cursors, as well as transient shapes (*e.g.* "zoom animation".)

Obviously, this list is not exhaustive; other layers can be found for specific applications. It does, however, apply to many applications (MacDraw, Illustrator, Idraw, *etc.*)
The model can manage many visible surfaces (multi-views): each virtual surface can be seen by any number of visible surfaces and each visible surface can display a virtual surface at any level.
Hence, the selection in a given view can differ from the selection in another view. The constraint grid can appear in one view and not in another, and be different in a third. A view can use a visible surface as a model whereas another uses it as the application data layer. See the example application at the end of the paper.

The Transformation Model

As explained in the previous section, the display of a virtual surface on a visible surface is done with a transformation and

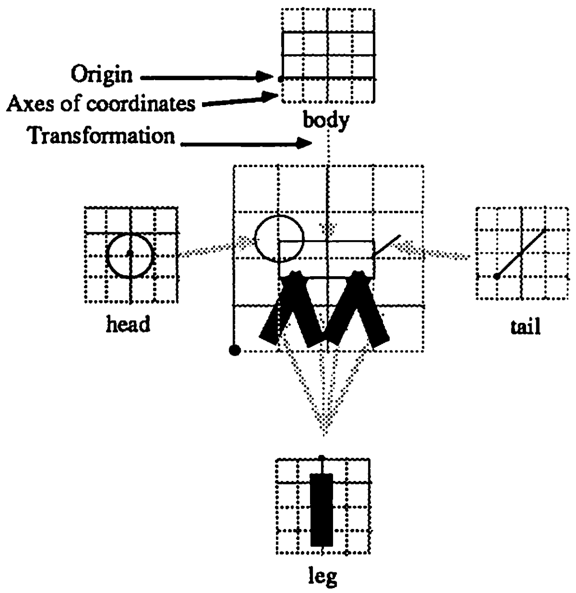


Figure 4: composition of structured graphic objects.

through a clip. The graphic structure is displayed on the visible surface analogeously to the way structured graphics packages (GKS [6], PHIGS [7]) build complex objects: by composition of simple objects, geometrically transformed (see figure 4).
In the mono-surface model, the visible surface is responsible for scrolling and zooming in 2D, or changing the view point in 3D (these operations are sometimes called *non-semantic* manipulations.) The transformation is therefore associated to the visible surface.
Our model retains similar properties but requires that some objects are aware of the transformation applied to them and can manage it explicitly when being drawn on the visible surface. Specifically, objects can behave in three ways to redraw themselves with a specific transformation:
• use the transformation for all components uniformly,
• use the transformation for positioning but not for dimensions (lines width, handles width, ...),
• don't use the transformation at all.
By construction, the application data layer uses the transformation for all components, since the transformation model has been chosen for this purpose. The selection and grid objects use the second behavior; they usually draw lines using a constant width of one pixel. The third is sometimes used to present graduated rules or display information like locator coordinates.

Implementation Issues

With the mono-layered model, most toolkits use a "lazy" redisplay mechanism: when a part of the visible surface should be updated, a function is called for the visible surface with the region to repaint as an argument. The toolkit is then responsible for redrawing the region. This mechanism is used either because that part has been made visible or because the graphics under it have changed (*i.e.* an object is created, modified or removed.) In the latter case, the data structure is updated and, for each visible surface where the virtual surface appears, the region to update is cleared and the redisplay mechanism is triggered.

In order to keep the simplicity of this mechanism with the multi-layer model, graphic objects should be able to actively control their drawing process to bypass the standard visualizing transformation. As explained in the previous paragraph, the redisplay algorithm calls a drawing function with a region to draw as an argument. In our model, the visualization transformation should also be passed as a parameter to the drawing function of each layer.
It should be noted that when a graphic structure is modified and the redisplay algorithm is notified that an area should be updated, the area cannot be computed just once for the abstract surface and propagated to each visible surface where it appears, because, here again, the treatment of the transformation has to be taken into account for each visible surface in order to compute the exact area.
The lazy redisplay mechanism used by the MacIntosh toolbox, as well as InterViews, ET++ and Xtv, can be easilly adapted to our model.
Note that no assumtions are made about the graphic model used to draw in the application data layer. The multi-layered model can be used with bitmap graphics with an alpha value, enhanced bitmap graphics (supporting zoom), bitmap graphics with some structured primitives (QuickDraw [2], X), device-independant 2 1/2D painting (PostScript [1]) or 3D (PHIGS).

Handling of 3D

Note also that the model is, to some extent, suitable for interactive manipulation of 3D graphics. Most 3D editors use a "wire frame" representation of 3D structures. The vertices are displayed using a transformation and manipulated using some 2D or 3D locator. The virtual surface contains 3D objects (virtual volume would be a better name) and the visible surfaces present a projection — through a visualizing transformation — of the virtual volume. The list of layers given for a conventional 2D graphical application still applies for a 3D editor:
• *the background* usually displays a solid color (either black or grey),
• *the graphical constraints visualization layer* can display a 3D grid or a trajectory,
• *the application data layer* displays the 3D objects,
• *the selected objects layer* displays handles of selected vertices or objects,
• *the lexical operations representation layer* displays the representation of the input device (the projection of a 3D mouse for instance).
The model cannot be used when objects are edited with their hidden surfaces removed because the cursor and selection should be consistent with the 3D structure. This case is, however, very unusual.

Handling of Events

A major benefit of multi-layer graphics is the simplicity of interaction handling. Each layer handles only those events for which it has expressed interests; control is distributed. When it receives an event, a layer can either:
• ignore it and pass it to layers below,
• handle it,
• handle it, then pass it to layers below,
• handle it, transform it and then pass it to layers below.
For example, figure 5 describes how events are handled by each layer if the selection mode works as follows:

Layer	Event	Behavior
Lexical:	PointerMoveDown	If a selection rectangle exists on the lexical layer, set its moving corner to the mouse position. Else, pass the event.
	PointerUp	If a rectangle exists on the lexical layer, delete the rectangle, then, pass a new event called SelectRect containing the rectagle boundings. Else, pass the event.
Selection:	PointerMoveDown	If the selection was being moved, move it with the pointer. If it was not moving and a ghost is under the pointer, move it with the pointer. Else, pass the event.
Data:	PointerDown	If an object is under the pointer, select it (create a ghost for it). Else, pass the event.
	SelectRect	Select the objects inside the rectangle.
Background:	PointerDown	Clear the selection, then, create a rectangle on the lexical layer at the mouse position.

Figure 5: description of the **selection**

- When the mouse is clicked outside any object, the selection is cleared.
- If the mouse is then dragged, a selection rectangle is drawn and follows the mouse.
- If the mouse button is released, objects inside the rectangle are selected and the rectangle is deleted.
- When the mouse is clicked on a graphic object, it becomes selected.
- If the mouse is clicked over a selected object and then dragged, a ghost[1] of the selection follows the pointer.
- When the mouse buttons are released, the ghosts are deleted and the objects they represent are moved to the new position.

Figure 5 also shows the declarative aspect of event handling with our model. Note that the selection rectangle is created by the background layer, because it is a default action, performed when no other layer handles the event more specifically.

For exotic devices, the lexical layer can also represent more sophisticated feedback than the traditional cursor. In our animation program for instance, we use a pressure sensitive wireless Wacom digitizer where the hot spot is represented as a cross and the pressure as a circle, with a radius proportional to the pressure. A sketching program using the digitizer to draw like a pencil would simply add some actions to the layers above (see figure 6).

[1] We call **ghost** a transient graphic object which represents the selection of a graphic object within the main data layer.

Layer	Event	Behavior
Lexical:	DigitMoveUp	Draw a cross (in Xor) centered at the hot spot and pass the event.
	DigitDown	Draw the cross and put the position and the pressure in a new list.
	DigitMoveDown	Erase the cross, draw a line from the previous point to the current point using the current pressure to compute its width, draw the cross at the current point and add the point and pressure to the list. Pass the event.
	DigitUp	Erase the cross, mark the area where the lines were traced as an area to redisplay, then, pass a new event called DigitTrace containing the list.
Data:	DigitTrace	Create an object with the list of points and select it.

Figure 6: adding actions for a pressure sensitive digitizer.

Logical Status of Layers

Most toolkits use a model inspired from the Smalltalk Model-View-Controller [8] (MVC) model: ET++ uses MVC, InterViews calls it Subject-View, Garnet uses a one-way constraint system. The idea of these models is to distinguish between object and representation and to provide a mechanism to keep them coherent. The PAC (Presentation, Abstraction, Controller) model of [4] is a generalization of this concept of separation. By using it, we can further clarify the notion of layer in our model.

In graphical applications, some abstract structure is to be displayed (see fig. 2). The graphic structure is considered as the presentation of the abstract structure. The graphic controller is responsible for keeping the graphical structure coherent with the abstract structure. It is also responsible for receiving input events and handling them according to the graphic semantics of the application. In most interactive graphic applications, the controller of this layer performs only "hit detection", that is, receives input events and determines which object they designate. The manipulation – either direct or through menus – is usually performed on selected objects.

With our model, we can describe the relationship between the graphic structure and the representation of its selection (its ghost) by a similar process (see fig. 7). The handles representing the selection are a presentation of the state "being selected" of the graphic structure.

As shown in the previous section, the ghost appears in the selected objects layer. In our model, the ghost is considered as a view of the graphical data structure representing the object in the main data layer. Most of the time, only geometric

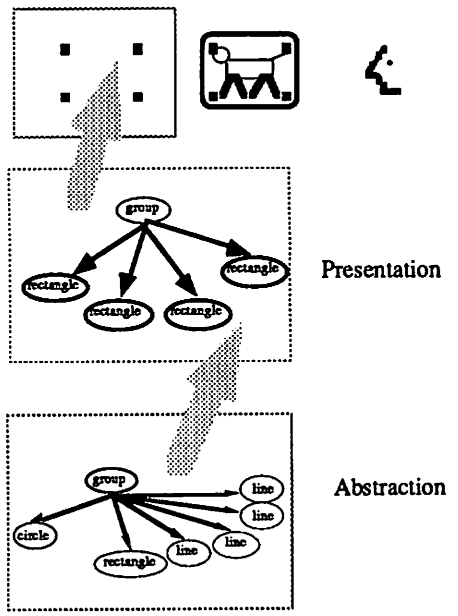


Figure 7: relationship between a graphic and its selection.

information is required to draw a ghost. The graphical aspect of the ghost can therefore be defined independantly of the application. Of course, in an object-oriented implementation, the ghost may be specialized to display some application-specific information.
The controller is responsible for maintaining the graphical coherence between the graphic structure of the main data layer and its ghost. It also interprets input events and handles direct manipulation. During direct manipulation, the controller maintains a corresponding graphic echo and can try to affect the manipulation by constraining input events (*e.g.* mouse move). The constraints here can be either lexical, syntactic or semantic, depending mainly on performance issues. Lexical constraints (like grid alignment constraints) are independant of the graphic data structures and of the semantics of the abstract data structure beeing manipulated. Syntactic constraints (like non-overlapping constraints) depend on the graphical structure. Semantic constraints (like hilighting only valid targets when interactively connecting two components of a graphic structure) depend on the graphic structure and the semantic structure.
Once the manipulation is terminated, the controller can modify the abstract data structure or ignore the effects if the manipulation is not considered valid.
This above structuration shows a recursive PAC organization (see figure 8).
The PAC model is also applicable to the graphical constraints visualization layer, which displays a presentation of some constraints. For example, if we use a simple grid alignment, the abstaction is composed of four values: the grid spacing and the offset from the origin. From this abstraction, the grid can be displayed. The abstraction is then used during the direct manipulation phase to align the input device events to grid values (see figure 9).
Finally, with the PAC model, the lexical operations representation layer displays some presentation of device states (the mouse cursor, for instance).
Each layer contains the presentation and handles the interaction for a specific category of abstractions.

Example of Use in an Application

With the multi-layer model, we have built a computer aided cartoon animation editor. The Unidraw toolkit, which offers a rich set of objects, has been modified for this purpose.
An animator can use both a mouse and a pressure sensitive Wacom stylus with multiple levels of pressure. The layers it uses are shown in figure 10.

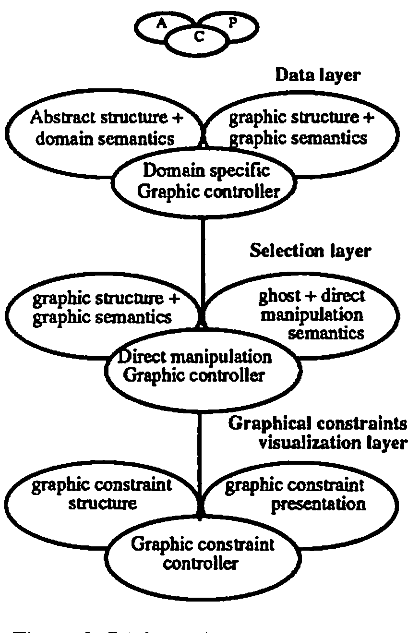

A
C
P
Data layer
Abstract structure + domain semantics
graphic structure + graphic semantics
Domain specific Graphic controller
Selection layer
graphic structure + graphic semantics
ghost + direct manipulation semantics
Direct manipulation Graphic controller
Graphical constraints visualization layer
graphic constraint structure
graphic constraint presentation
Graphic constraint controller

Figure 8: PAC relations between layers.

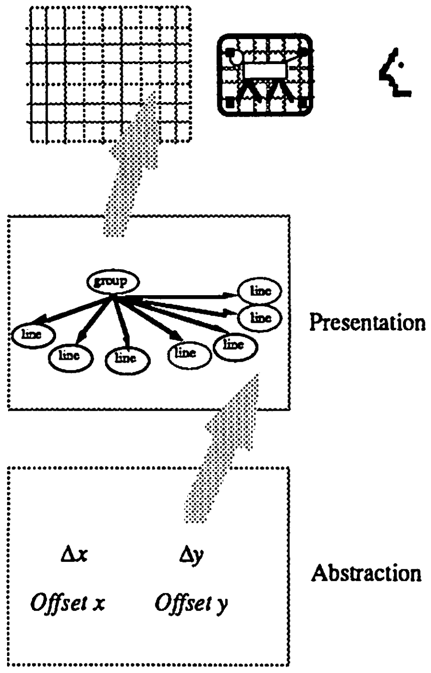


Figure 9: PAC model for a grid alignment constraint.

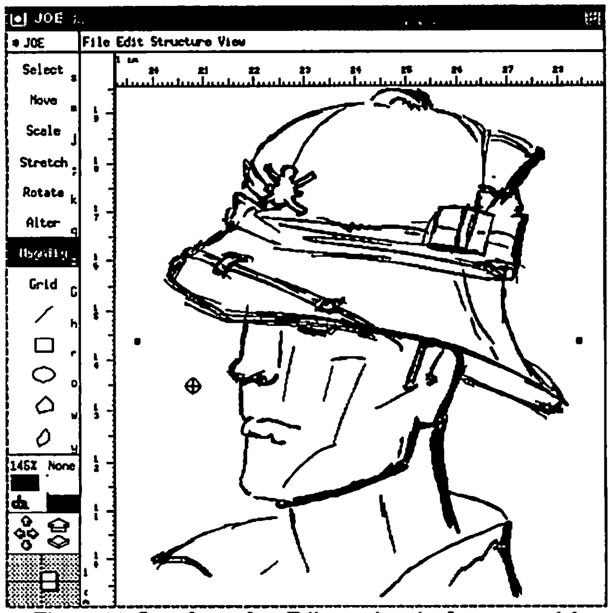


Figure 11: Interface of an Editor using the Layers model.

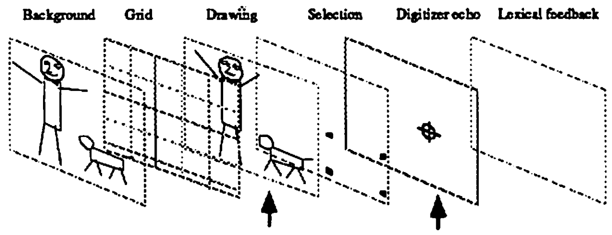


Figure 10: layers of a 2D animation graphic editor.

Compared with a conventional 2D graphic editor, we have added the *Digitizer echo layer* which displays a cross centered at the hot spot of the stylus, with a width equal to the selected drawing line width. When the stylus touches the digitizer, a circle appears with a radius proportional to the pressure applied, as shown in figure 11.
The graphic editor can show multiple views of the same drawing (in the main data layer). Each view also contains the Digitizer echo layer. This feature is useful when working on a small area of a drawing, which is usually zoomed on one view but not on another. The cursor provides the visual insurance that the two views display the same drawing.
The background layer can contain other drawings, greyed out. This feature is meant to emulate the use of tracing paper.

Related Work

The multi-layer model is not completely new; previous systems have used multiple layers for graphic output or event handling. However, no model generalizes this notion as ours does.
Some window systems offer support for stacked windows, either for graphic output or for event handling. NeWS [13] has a special type of window called "overlay canvas" which is used to display transient data. Overlay canvases optimize the redisplay by relaxing the drawing model of PostScript. They usually use either an overlay plane of the screen on the root window when the hardware offers such a device, or draws all the shapes using Xor raster operations. Overlay canvases offer just one level of layering, which is a strong limitation.
The X window system offers transparent input-only windows for event handling. As their name implies, no graphic output can be done on such windows.
The HyperCard [5] system has a two layer model for handling graphics and events, which is very close to our model. However, HyperCard is not extensible and can not be considered as a complete toolkit.
It is also interesting to notice that some real devices do provide multi-layered graphics, like heads-up displays on military planes.

Future work

We are currently working on a graphic editor for designing interactive graphical applications, using this model to express event handling graphically.

Conclusion

The multi-layered multi-view model simplifies both the management of graphical output and the description of event handling for interactive graphical applications. It clarifies the realization of objects appearing in a graphical application, like selection, cursor, grid or selection rectangle. It also permits a clear description of event handling for a variety of input devices. We believe this model can unify the handling of a variety of problems that are currently solved with ad hoc approaches.

Acknowledgments

The model described in this article is an extension to the model used in the Xtv toolkit developped at the University of Orsay Paris-Sud by the Human-Interface group of the LRI. Thanks to all its authors. Thanks also to Michel Beaudouin-Lafon, Stéphane Chatty and Chris Weikart for their comments and suggestions.

Bibliography

[1] Adobe, PostScript Language Reference Manual, Addison Weseley, Reading Mass., 1985.
[2] Apple Computer, Inside Macintosh, Volume I, Addison Weseley, Reading Mass., 1986.
[3] M. Beaudouin-Lafon, Y. Berteaud, S. Chatty, Creating Direct Manipulation Applications with Xtv, Proc. European X Window Conference (EX), Nov. 1990.
[4] J. Coutaz, Interface Homme-Ordinateur : Conception et Réalisation, Dunod, 1990.
[5] G. Harvey, Understanding HyperCard for Version 1.1, Sybex Books Publishers, 1988.
[6] International Organization for Standardization, Information processing systems – Computer Graphics – Graphical Kernel System (GKS) functional description, ISO IS 7942, July, 1985.
[7] International Organization for Standardization, Information processing systems – Computer Graphics – Programmer's Hierarchical Interface to Graphics (PHIGS) functional description, ISO DP 9592, October 1986.
[8] G. Krasner, S. Pope, A Cookbook for using the Model-View-Controller user interface paradigm in Smalltalk-80, JOOP, August/September 1988, pp. 26—49.
[9] M. A. Linton, J. M. Vlissides and P. R. Calder, Composing User Interfaces with InterViews. IEEE Computer, February 1989, pp. 8–22.

[10] X Toolkit Library – C Language Interface, X protocol Version 11, MIT, 1987.

[11] B. A. Myers *et al*, Garnet : Comprehensive Support for Graphical, Highly Interactive User Interface, IEEE Computer, November 1990, pp. 71–85.

[12] R. W. Scheifler, J. Gettys, The X Window System, ACM Transactions on Graphics 5(2), April 1986, pp. 79–109.

[13] SUN Microsystems Inc. : NeWS Manual ; SUN Microsystems Inc., 2250 Garcia Avenue, Mountain View, CA 94043.

[14] J. M. Vlissides, M. A. Linton, Unidraw: A Framework for Building Domain-Specific Graphical Editors, ACM-Transactions on Information Systems, 8, 3, July 1990, pp. 237–269.

[15] A. Weinand, E. Gamma, R. Marty, ET++ — An Object-Oriented Application Framework in C++, in ACM-OOPSLA'88 proceedings, San-Diego, SIGPLAN Notices, 23, 11, November, 1988, pp. 46–57.

A Linear Constraint Technology for Interactive Graphic Systems

Richard Helm, Tien Huynh, Catherine Lassez, Kim Marriott

I.B.M. Thomas J. Watson Research Center,
P.O. Box 704, Yorktown Heights, NY 10598

Abstract

Constraints provide a natural formalism for user-interface design and graphical layout. Recent results and algorithms from symbolic computation and geometry provide new techniques to manipulate linear arithmetic constraints. We show how these results can be applied to interactive graphical user-interfaces and how they extend the capabilities of previous interactive constraint-based user interface systems. We propose an architecture for such systems based on these techniques.

Keywords: Linear Arithmetic Constraints, Interactive Techniques

1 Introduction

Constraints are a natural formalism for specifying user-interface design and graphical layout. They have shown their utility in interactive systems [1, 2, 3, 11, 12, 13, 14, 15], in graphic specification languages [5, 17] and recently in visual language parsing [4].

The requirements of adequate response time and graphical feedback for interactive systems demand certain capabilities of constraint solvers. In particular, previous work, for example ThingLab and ThingLab II [2, 11] suggests that constraint solvers for interactive graphics must provide:

- Incremental addition and deletion of constraints.
- Fast generation of plans of execution when the object that is the focus of manipulation changes.
- Adequate feedback bandwidth when manipulating a graphic object.

Other capabilities which constraints can provide and which an interactive constraint-based system should support are:

- Determining and presenting the range of values that a variable can take. For example, if a point is going to be dragged around on the screen, the system should be able to present graphically to the user whereabouts this point can be moved.
- Support the definition and compilation of compound objects. Although, this is straightforward in systems without constraints, the addition of constraints raises new issues. In particular, efficiently representing the constraints in the compound object, and determining which variables of a compound graphic object define all objects it contains.

In addition the constraint solver must support other, more usual, operations associated with constraints. These include:

- Detecting that a system of constraints is unsatisfiable, and identifying which constraints must be removed to restore satisfiability.
- Detecting an under-constrained system and identifying which variables must be further constrained.

Both the constraints causing unsatisfiability, and variables that need to be further constrained must be indicated to the user.

Recent results and algorithms from symbolic computation [6, 8, 9] provide powerful techniques to manipulate sets of linear arithmetic constraints containing both equalities and inequalities. We show that when applied to interactive constraint-based user-interface systems, these techniques give new capabilities, and enhance interaction through improved user-feedback. In particular, they provide:

1. A *canonical form* for a set of linear arithmetic constraints. The canonical form is concise, does not contain redundant constraints, identifies the degrees of freedom in the constraints, and makes explicit the equalities implied by the constraints.
   There is an incremental algorithm to compute the canonical form. This algorithm also determines whether or not the constraints are satisfiable. The canonical form is a key for both efficient representation and manipulation of constraints.

2. A *parametric solved form* for the solutions of a set of constraints in terms of distinguished parametric variables. The solved form, which corresponds to the *plans* of Thinglab II, allows the values of variables to be computed rapidly from the parameters. This permits the rapid re-satisfaction of the constraints when objects are being manipulated by the user.
3. An efficient projection algorithm to compute the range of values of distinguished variables. This allows the user to get feedback about the region within which selected objects on the screen can be moved while still satisfying the constraints. Projection also plays a role in computing the manipulable interface of compound objects.
4. Techniques to deal with unsolvable sets of constraints by identifying minimal unsolvable subsets. These provide feedback as to which constraints need to be relaxed or removed to restore solvability.
5. Techniques to determine if a set of constraints is underconstrained, that is if some variables cannot be uniquely determined in terms of certain distinguished parameters, and consequently which variables must be further constrained.

This work extends previous constraint technology for interactive constraint-based systems in two ways.

First, we present new feedback mechanisms and interaction styles for constraint-based systems. These are based on new techniques for extracting information about causes of over- and under- constrainedness in sets of constraints, and determining relationships on specific variables implied by the constraints. These techniques rely on manipulating constraints symbolically. To our knowledge these features have not appeared in previous systems.

Secondly, we extend previous work that uses symbolic techniques to solve constraints by allowing simultaneous linear equations and inequalities. This extension is important because such constraints arise naturally when specifying graphical layout. For efficiency reasons, previous systems have dealt mainly with systems of acyclic linear equations [3, 11, 15], and solved these constraints using local propagation techniques. Recent research has addressed the efficient and incremental recompilation of these types of constraints in response to user interaction [11]. For the more general constraints considered here, however, local propagation is not sufficient; global techniques must be used. We note that Witkin [16] and Nelson [12] deal with more powerful constraints, but use numerical techniques for constraint satisfaction. It is not clear how the feedback mechanisms we present can be provided using these numerical techniques.

The rest of the paper is organized as follows. In the next section, we present an example of a typical interactive session which illustrates these new techniques. In Section 3, we discuss two key elements of the underlying constraints technology: a canonical representation for constraint, and a new projection algorithm particularly suited to this application. In Section 4, we propose an architecture for a constraint manipulation sub-system to be used within an interactive user interface system. In Section 5 we present some empirical results concerning the performance of this constraint technology.

2 Example

To illustrate some of the capabilities of the constraint technology, we present a simple hypothetical session with an interactive constraint-based editor.

Consider the diagram illustrated in Figure 1 which consists of two pieces of text $T1$ and $T2$, two rectangles $R1$ and $R2$ and a surrounding box B. Suppose that the user wishes to satisfy the following requirements.

1. $R1$ and $R2$ have a fixed aspect ratio.
2. $R1$ and $R2$ have the same size and cannot overlap.
3. Ti is centered in Ri and Ri is large enough to contain Ti.
4. B contains both $R1$ and $R2$, and they are "nicely" placed inside B, that is the rectangles are equidistant from the borders of B and from each other.

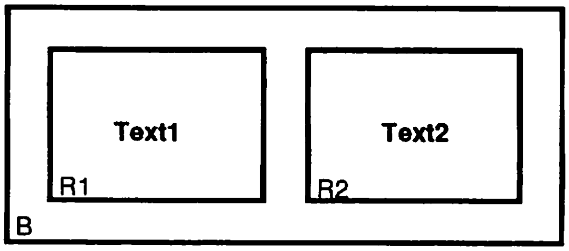


Figure 1: Layout of the diagram

To create this diagram, the user adds and deletes graphic objects and constraints and modifies the parameters (or attributes) of the objects. The resulting layout is defined by a set of constraints. These fall into the following categories: *local* constraints, which express the relationships between the modifiable parameters of system-defined objects; *global* constraints, provided by the user, which express the relations between objects; and *anchor* constraints which express that certain points or attributes are fixed.

Typically, local constraints are pre-defined for each object, and are highly redundant to allow multiple ways to define an object. In our example, each rectangle may be defined by its opposite vertices or its center and a vertex. Thus each rectangle has as parameters, its four vertices - bl, br, ul, ur for bottom-left, bottom right, etc...; its center point c, and its extent e in x and y dimensions. The local constraints for an object O over these parameters are

$$\begin{array}{ll} O_{cx} = 0.5 * O_{blx} + 0.5 * O_{brx} & O_{bry} = O_{bly} \\ O_{cy} = 0.5 * O_{bly} + 0.5 * O_{uly} & O_{ulx} = O_{blx} \\ O_{brx} = O_{blx} + O_{ex} & O_{urx} = O_{brx} \\ O_{uly} = O_{bly} + O_{ey} & O_{ury} = O_{uly} \\ O_{ex} \geq 0 & O_{ey} \geq 0. \end{array}$$

Global constraints are provided by the user and are defined over the parameters of the relevant objects. For example, the requirement that $T1$ is contained in $R1$ is expressed by

$$\begin{array}{l} T1_{blx} \geq R1_{blx} \\ T1_{bly} \geq R1_{bly} \\ T1_{urx} \leq R1_{urx} \\ T1_{ury} \leq R1_{ury}. \end{array}$$

The entire set of local and global constraints which capture the previous 4 requirements for Figure 1 is shown in Figure 5. The large number of constraints generated by this relatively simple example is typical – systems such as ThingLabII generate similar numbers of constraints for similarly sized examples. The large numbers of constraints means that efficient representation and manipulation of constraints is important.

Anchor constraints are equalities which fix the values of variables of objects. For instance, the constraint

$$B_{blx} = B_{bly} = 0$$

expresses that the lower left corner of B is anchored at the origin of the screen. Anchor constraints are added by the user when the attributes of objects are not to be modified.

When creating this diagram, the user adds and deletes graphic objects and constraints, and modifies the parameters (or *attributes*) of the objects. As constraints and objects are added, the solvability of the entire set must be checked. If it becomes unsatisfiable, information as to which constraints need to be modified to restore solvability is fed back to the user by highlighting a graphical representation of the offending constraints.

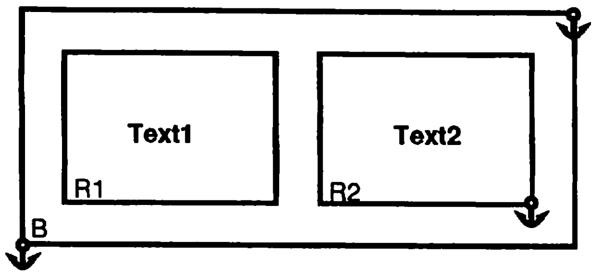


Figure 2: Three anchor points were selected

During a session, as graphic objects are moved around and their attributes modified, the system gives feedback about the current constraints and presents this in a suitable graphical format. For instance, suppose the user wants to move the upper-right corner of $R1$. Initially the vertices of B, one vertex of $R2$ and the text size are "anchored" so that the system will not change their present values (see Figure 2). At this point the set of constraints is over-constrained. When the user selects the upper-right corner of $R1$, the system indicates that $R1$ cannot be moved. This is because the constraints imply that the coordinates of $R1$ are fixed. Note that this information is not explicit in the original constraints.

The system can then indicate which anchor constraints need to be removed to restore some degrees of freedom in the system. If the user now removes the anchor-constraint on $R2$, it is now possible to automatically infer, from the constraints, the possible values for $R1$. The possible values are given by the constraints

$$\begin{array}{c} R1_{urx} - \frac{4}{3} R1_{ury} = 0 \\ 110 \leq R1_{ury} \leq 150 \end{array}$$

defining the line segment as depicted in Figure 3. The system displays this line and the cursor is constrained to remain on it. Whenever the cursor is moved, the display is updated to reflect the new configuration (Figure 4).

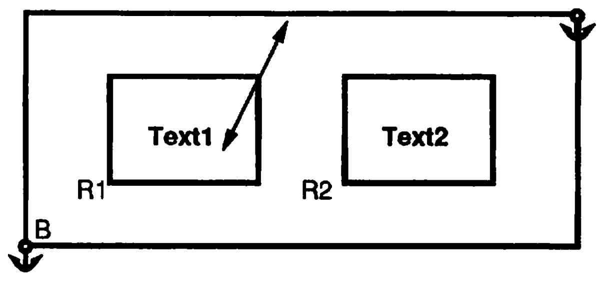


Figure 3: Range of motion for point $R1_{ur}$

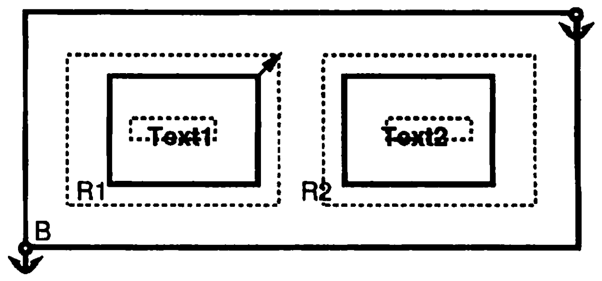


Figure 4: Display reflecting motion of point $R1_{ur}$

3 The Linear Constraint Technology

From the previous example, we can see that an underlying constraint technology for interactive constraint-based systems should provide efficient:

- Incremental addition and deletion of constraints.
- Detection of unsatisfiability and identification of which constraints or anchor constraints are causing it.
- Rapid re-satisfaction of existing constraints when a small number of parameters, such as the location of a vertex, are changed.
- Detection of under-constrained system and identification of which parameters can be fixed to constrain it.
- Recognition of an over-constrained constraint system and identification of the responsible anchor constraints.
- Computation of the range of values that a parameter can take while leaving the system satisfiable.

Recent results from symbolic computation provide a technology for linear arithmetic constraints with these capabilities. The key to an efficient implementation is based on a new representation for sets of linear constraints called the *canonical form* [10], and a new algorithm for variable projection [9]. We now discuss the canonical form, how it supports the compilation of plans of execution or parametric solved forms, and projection.

The Canonical Form

The sets of constraints that arise in interactive systems often contain redundancy. Local constraints defining attributes of objects often contain redundancy to allow flexible definition of objects. Constraints on objects can become redundant as a user adds further constraints. Because of the potentially large number of constraints that can be generated in interactive systems, it is important to have non-redundant representations. In the technology we present, this representation is given by the canonical form.

The *canonical form*[1] of a set of linear arithmetic constraints consists of:

1. A set of equations that defines the affine hull of the solution set of the constraints. This affine hull is the space having the smallest dimension that contains the set of solutions to the constraints.
2. A set of inequalities that define the full dimensional solution set.

The canonical form has the following important properties: it contains no redundant constraints; it identifies the degrees of freedom in the constraints; and it makes explicit the equalities implied by the constraints.

[1]Note: the full definition of the canonical form also includes negative constraints which are not discussed here. The interested reader is referred to the reference for a complete treatment.

Eliminating redundancy is important because typically the system of constraints may contain many redundant constraints mainly due to the local constraints. For some systems of constraints, the corresponding canonical form has an order of magnitude fewer constraints. Making equalities explicit is important because they can greatly simplify the set of constraints. The degrees of freedom of the set of constraints is given by the number of variables less the number of equalities in the canonical form. This means it is possible to determine if the system is under-constrained and which variables need to be further constrained.

Transforming an arbitrary set of constraints into its canonical form is a complex three-stage process using a quasi-dual formulation of optimization techniques (such as the simplex method) from Linear Programming [7, 10]. The three stages are identification of implicit equalities, simplification, and elimination of redundancy. The first stage performs a test for satisfiability. If the system of constraints is unsatisfiable, then a minimal subset of constraints causing unsatisfiability is identified.

Computing the canonical form from scratch is expensive. However, we use an incremental algorithm that efficiently recomputes the canonical form when constraints are added. This is an important consideration when interactively constructing systems of constraints.

The system given in Figure 5 has the canonical form given in Figure 6. The original system has 54 equalities, and 28 inequalities involving 32 variables. The canonical form has 51 equalities, and 10 inequalities involving 9 variables. Note the substantially reduced number of variables in the inequalities.

The Parametric Solved Form

One of the most important operations in an interactive graphics system is the manipulation of objects on the display. To do this efficiently requires computing a plan of execution in terms of the object being manipulated, and then continually re-executing this plan.

In our technology, a plan of execution is called a *parametric solved form with respect to a set of parametric variables*. The parametric solved form is a set of constraints such that all dependent variables are expressed in terms of the parameters, and only the parametric variables occur in the inequalities. The parametric variables are those that correspond to the object being manipulated. Thus the parametric solved form has the property that if the constraints contain the variables $x_1, .., x_n$, and the parameters are $x_1, ..., x_i$, then for $j = i+1, ..., n$, there exists an f_j such that $x_j = f_j(x_1, ..., x_i)$.

The system of constraints corresponding to Figure 4 when parameterized by variable $R1_{ury}$ has the parametric solved form given in Figure 7. This solved form consists of 59

$$\begin{array}{llll}
T1_{cx} = 0.5 * T1_{blx} + 0.5 * T1_{brx} & & T2_{cx} = 0.5 * T2_{blx} + 0.5 * T2_{brx} & \\
T1_{cy} = 0.5 * T1_{bly} + 0.5 * T1_{uly} & & T2_{cy} = 0.5 * T2_{bly} + 0.5 * T2_{uly} & \\
T1_{brx} = T1_{blx} + T1_{ex} & T2_{brx} = T2_{blx} + T2_{ex} & T1_{uly} = T1_{bly} + T1_{ey} & T2_{uly} = T2_{bly} + T2_{ey} \\
T1_{bry} = T1_{bly} & T1_{ulx} = T1_{blx} & T2_{bry} = T2_{bly} & T2_{ulx} = T2_{blx} \\
T1_{urx} = T1_{brx} & T1_{ury} = T1_{uly} & T2_{urx} = T2_{brx} & T2_{ury} = T2_{uly} \\
T1_{blx} \geq R1_{blx} & T1_{bly} \geq R1_{bly} & T2_{blx} \geq R2_{blx} & T2_{bly} \geq R2_{bly} \\
T1_{urx} \leq R1_{urx} & T1_{ury} \leq R1_{ury} & T2_{urx} \leq R2_{urx} & T2_{ury} \leq R2_{ury} \\
T1_{cx} = R1_{cx} & T1_{cy} = R1_{cy} & T2_{cx} = R2_{cx} & T2_{cy} = R2_{cy} \\
T1_{ex} \geq 0 & T1_{ey} \geq 0 & T2_{ex} \geq 0 & T2_{ey} \geq 0 \\
R1_{cx} = 0.5 * R1_{blx} + 0.5 * R1_{brx} & & R2_{cx} = 0.5 * R2_{blx} + 0.5 * R2_{brx} & \\
R1_{cy} = 0.5 * R1_{bly} + 0.5 * R1_{uly} & & R2_{cy} = 0.5 * R2_{bly} + 0.5 * R2_{uly} & \\
R1_{brx} = R1_{blx} + R1_{ex} & R2_{brx} = R2_{blx} + R2_{ex} & R1_{uly} = R1_{bly} + R1_{ey} & R2_{uly} = R2_{bly} + R2_{ey} \\
R1_{bry} = R1_{bly} & R1_{ulx} = R1_{blx} & R2_{bry} = R2_{bly} & R2_{ulx} = R2_{blx} \\
R1_{urx} = R1_{brx} & R1_{ury} = R1_{uly} & R2_{urx} = R2_{brx} & R2_{ury} = R2_{uly} \\
R1_{blx} \geq B_{blx} & R1_{bly} \geq B_{bly} & R2_{blx} \geq B_{blx} & R2_{bly} \geq B_{bly} \\
R1_{urx} \leq B_{urx} & R1_{ury} \leq B_{ury} & R2_{urx} \leq B_{urx} & R2_{ury} \leq B_{ury} \\
R1_{ex} = 2 * R1_{ey} & R2_{ex} = 2 * R2_{ey} & & \\
R1_{ex} \geq 0 & R1_{ey} \geq 0 & R2_{ex} \geq 0 & R2_{ey} \geq 0 \\
B_{brx} = B_{blx} + B_{ex} & B_{bry} = B_{bly} & B_{urx} = B_{brx} & B_{ury} = B_{uly} \\
B_{ulx} = B_{blx} & B_{uly} = B_{bly} + B_{ey} & B_{ex} \geq 0 & B_{ey} \geq 0 \\
M_x = R1_{blx} - B_{blx} & M_x = R2_{blx} - R1_{brx} & M_y = R1_{bly} - B_{bly} & M_y = B_{uly} - R1_{uly} \\
M_x = B_{brx} - R2_{brx} & M_y = R2_{bly} - B_{bly} & M_y = B_{uly} - R2_{uly} & M_x \geq 0,\ M_y \geq 0 \\
3 * M_x + R1_{ex} + R2_{ex} - B_{ex} = 0 & & 2 * M_y + R1_{ey} - B_{ey} = 0 & R1_{ey} - R2_{ey} = 0
\end{array}$$

Figure 5: Original Set of 82 Constraints

equations and the two inequalities

$$110 \leq R1_{ury} \qquad R1_{ury} \leq 150.$$

One can readily see how all variables are either assigned to constants or are expressed solely in terms of $R1_{ury}$. Thus, only one degree of freedom remains in the solved form.

The parametric solved form is efficiently derived from the canonical form (see Section 5). It can then be either interpreted directly or used to compile efficient code to compute the values of dependent variables as the values of parameters are changed.

Projection

Projection provides a technique to examine the relationships between particular variables that are implied by the constraints. Projecting onto a single variable gives the range of values it can take while still satisfying the constraints. Projecting a system of constraints onto a set of variables shows how these variables are inter-related in any solution to the constraints.

In the example in Section 2, we saw how projecting the constraints on to variables R_{urx}, R_{ury} indicated that they were constrained to be on the line $R1_{urx} - \frac{4}{3}R1_{ury} = 0$ where $110 \leq R1_{ury} \leq 150$.

Unfortunately, the doubly-exponential complexity of general algorithms for projection has prevented its use in many application domains. Recently, however, a new algorithm has been developed [9] that is very efficient when the number of variables in the projection space is small. This is exactly the case we are interested in, as we typically project onto a small number of variables (typically one or two) corresponding to the object currently being manipulated.

The new algorithm computes a projection by successive approximations using an on-line algorithm for convex hull construction in the projection space. It provides an exact solution when the size of the output is small, and an approximation (upper or lower) when the size of the output is unmanageable. Previous methods usually failed to produce any output, even in small cases, because of the enormous amount of intermediate computation. Initial testing has shown extremely good performance, especially for small projection spaces.

4 Proposed Architecture

In this section we propose an architecture for a constraint manipulation sub-system within an interactive constraint-based user-interface system. The architecture exploits the technology described in the previous section. This architecture must address issues of incrementality, interaction latency and feedback bandwidth [11]. These issues are most critical when anchor constraints and the values of the parameters of objects are changed during manipulation.

To address these issues, we use a two level architecture which maintains sets of constraints in canonical form. The first, the *free canonical form*(FCF), is a canonical form of the local and global constraints. The second, the *anchored canonical form*(ACF), is a canonical form for the entire set of local, global and anchor constraints. In addition, local constraints associated with primitive and compound objects are kept in canonical form.

The FCF does not change often, only when local or global constraints are added or deleted. However, whenever an

$$\begin{aligned}
T1_{cx} &= B_{ulx} + B_{ey} + M_x - 2M_y \\
T1_{blx} &= -T1_{urx} + 2B_{ulx} + 2B_{ey} + 2M_x - 4M_y \\
T1_{brx} &= T1_{urx} \\
T1_{ex} &= 2T1_{urx} - 2B_{ulx} - 2B_{ey} - 2M_x + 4M_y \\
T1_{ulx} &= -T1_{urx} + 2B_{ulx} + 2B_{ey} + 2M_x - 4M_y \\
T2_{cx} &= B_{ulx} + 3B_{ey} + 2M_x - 6M_y \\
T2_{blx} &= -T2_{urx} + 2B_{ulx} + 6B_{ey} + 4M_x - 12M_y \\
T2_{brx} &= T2_{urx} \\
T2_{ex} &= 2T2_{urx} - 2B_{ulx} - 6B_{ey} - 4M_x + 12M_y \\
T2_{ulx} &= -T2_{urx} + 2B_{ulx} + 6B_{ey} + 4M_x - 12M_y \\
R1_{blx} &= B_{ulx} + M_x \\
R1_{cx} &= B_{ulx} + B_{ey} + M_x - 2M_y \\
R1_{brx} &= B_{ulx} + 2B_{ey} + M_x - 4M_y \\
R1_{urx} &= B_{ulx} + 2B_{ey} + M_x - 4M_y \\
R1_{ex} &= 2B_{ey} - 4M_y \\
R2_{bly} &= B_{uly} - B_{ey} + M_y \\
R2_{cy} &= B_{uly} - 0.5B_{ey} \\
R2_{uly} &= B_{uly} - M_y \\
R2_{ury} &= B_{uly} - M_y \\
B_{blx} &= B_{ulx} \\
B_{brx} &= B_{ulx} + 4B_{ey} + 3M_x - 8M_y \\
B_{urx} &= B_{ulx} + 4B_{ey} + 3M_x - 8M_y \\
R1_{ulx} &= B_{ulx} + M_x \\
R2_{ulx} &= B_{ulx} + 2B_{ey} + 2M_x - 4M_y \\
R2_{ey} &= B_{ey} - 2M_y \\
B_{ex} &= 4B_{ey} + 3M_x - 8M_y
\end{aligned}$$

$$\begin{aligned}
T1_{cy} &= B_{uly} - 0.5B_{ey} \\
T1_{bly} &= -T1_{ury} + 2B_{uly} - B_{ey} \\
T1_{uly} &= T1_{ury} \\
T1_{bry} &= -T1_{ury} + 2B_{uly} - B_{ey} \\
T1_{ey} &= 2T1_{ury} - 2B_{uly} + B_{ey} \\
T2_{cy} &= B_{uly} - 0.5B_{ey} \\
T2_{bly} &= -T2_{ury} + 2B_{uly} - B_{ey} \\
T2_{uly} &= T2_{ury} \\
T2_{bry} &= -T2_{ury} + 2B_{uly} - B_{ey} \\
T2_{ey} &= 2T2_{ury} - 2B_{uly} + B_{ey} \\
R1_{bly} &= B_{uly} - B_{ey} + M_y \\
R1_{cy} &= B_{uly} - 0.5B_{ey} \\
R1_{uly} &= B_{uly} - M_y \\
R1_{ury} &= B_{uly} - M_y \\
R2_{blx} &= B_{ulx} + 2B_{ey} + 2M_x - 4M_y \\
R2_{cx} &= B_{ulx} + 3B_{ey} + 2M_x - 6M_y \\
R2_{brx} &= B_{ulx} + 4B_{ey} + 2M_x - 8M_y \\
R2_{urx} &= B_{ulx} + 4B_{ey} + 2M_x - 8M_y \\
R2_{ex} &= 2B_{ey} - 4M_y \\
B_{bly} &= B_{uly} - B_{ey} \\
B_{bry} &= B_{uly} - B_{ey} \\
B_{ury} &= B_{uly} \\
R1_{ey} &= B_{ey} - 2M_y \\
R1_{bry} &= B_{uly} - B_{ey} + M_y \\
R2_{bry} &= B_{uly} - B_{ey} + M_y
\end{aligned}$$

$$\begin{aligned}
&M_y + T1_{ury} - B_{uly} \le 0 \\
&-B_{ey} - 2T1_{ury} + 2B_{uly} \le 0 \\
&M_y - B_{uly} + T2_{ury} \le 0 \\
&-B_{ey} + 2B_{uly} - 2T2_{ury} \le 0 \\
&-M_x \le 0
\end{aligned}$$

$$\begin{aligned}
&-2T1_{urx} + 2B_{ulx} + 2B_{ey} + 2M_x - 4M_y \le 0 \\
&-B_{ulx} - 4B_{ey} - 2M_x + 8M_y + T2_{urx} \le 0 \\
&2B_{ulx} + 6B_{ey} + 4M_x - 12M_y - 2T2_{urx} \le 0 \\
&T1_{urx} - B_{ulx} - 2B_{ey} - M_x + 4M_y \le 0 \\
&-M_y \le 0
\end{aligned}$$

Figure 6: Canonical Form of Original Set – 61 constraints

anchor constraint is changed, or the picture is manipulated via a new object, a new ACF is incrementally computed from the current FCF. Obviously – it is better not to redo the work already done in finding redundancy and implicit equalities. The role of the ACF is to be a parametric solved form with respect to the variables in the object currently being manipulated. This allows efficient updating of the display when the parameters are changed.

When objects or global constraints are added to the system new canonical forms are computed from the old one. Firstly, an FCF for the new local and global constraints is computed incrementally from the old FCF. Then, the ACF for the entire system of constraints is computed incrementally from the new FCF by adding the anchor constraints. This is quite efficient because the anchor constraints are always equations. Computation of these canonical forms will reveal if the system is unsatisfiable, and if so which constraints are at fault.

Deletion of objects or global constraints is more problematic. However, because local constraints associated with each object are kept in canonical form, then a new FCF and subsequently a new ACF can be computed relatively quickly. Thus the architecture provides:

- Incremental addition and deletion of constraints.
- Detection of unsatisfiability and identification of which constraints are the cause.

When an object is selected for manipulation, the ACF is examined to see if it is in parametric solved form with respect to the variables in the object. There are three cases to consider: the ACF is under-constrained, over-constrained, or it is in parametric solved form.

Examination of the ACF will reveal the system is over-constrained if there are no degrees of freedom, that is all variables are uniquely determined. For instance, consider the anchor constraints in Figure 2. These constraints, as follows,

$$\begin{array}{lll}
B_{llx} = B_{lly} = 0 & B_{urx} = 400 & B_{ury} = 200 \\
T1_{ex} = 30 & T1_{ey} = 10 & T2_{ex} = 40 \\
T2_{ey} = 20 & R2_{lrx} = 300 & R2_{lry} = 87.5
\end{array}$$

will fix the width and height of the boxes containing text. Before adding these anchor constraints, the FCF is that shown in Figure 6. When the anchor constraints are added, we derive from the FCF, the following very simple ACF with 60 equations that assign each variable to a constant:

$$T1_{cy} = 100, \quad T1_{blx} = 110, \quad T1_{bly} = 95, \ldots$$

As every variable is uniquely determined, there are no remaining degrees of freedom. Thus, with these anchor

$$
\begin{array}{ll}
T1_{cx} = 200 - \frac{2}{3}R1_{ury} & T1_{cy} = 100 \\
T1_{blx} = 185 - \frac{2}{3}R1_{ury} & T1_{bly} = 95 \\
T1_{brx} = 215 - \frac{2}{3}R1_{ury} & T1_{ury} = 105 \\
T1_{ulx} = 185 - \frac{2}{3}R1_{ury} & T1_{bry} = 95 \\
T1_{ex} = 30 & T1_{ey} = 10 \\
T2_{cx} = 200 + \frac{2}{3}R1_{ury} & T2_{cy} = 100 \\
T2_{blx} = 180 + \frac{2}{3}R1_{ury} & T2_{bly} = 90 \\
T2_{brx} = 220 + \frac{2}{3}R1_{ury} & T2_{uly} = 110 \\
T2_{ulx} = 180 + \frac{2}{3}R1_{ury} & T2_{bry} = 90 \\
T2_{ex} = 40 & T2_{ey} = 20 \\
R1_{cx} = 200 - \frac{2}{3}R1_{ury} & R1_{cy} = 100 \\
R1_{blx} = 400 - \frac{8}{3}R1_{ury} & R1_{bly} = 200 - R1_{ury} \\
R1_{brx} = \frac{4}{3}R1_{ury} & R1_{bry} = 200 - R1_{ury} \\
R1_{ulx} = 400 - \frac{8}{3}R1_{ury} & R1_{uly} = -400 + 4R1_{ury} \\
R1_{ex} = -400 + 4R1_{ury} & R1_{ey} = -200 + 2R1_{ury} \\
R2_{blx} = 400 - \frac{4}{3}R1_{ury} & R2_{bly} = 200 - R1_{ury} \\
R2_{cx} = 200 - \frac{2}{3}R1_{ury} & R2_{cy} = 100 \\
R2_{brx} = \frac{8}{3}R1_{ury} & R2_{bry} = 200 - R1_{ury} \\
R2_{urx} = \frac{8}{3}R1_{ury} & R2_{ury} = R1_{ury} \\
R2_{ex} = -400 + 4R1_{ury} & R2_{ey} = 200 - R1_{ury} \\
B_{blx} = 0 & B_{bly} = 0 \\
B_{brx} = 400 & B_{bry} = 0 \\
B_{urx} = 400 & B_{ury} = 200 \\
B_{ulx} = 0 & B_{uly} = 200 \\
B_{ex} = 400 & B_{ey} = 200 \\
T1_{urx} = 400 - \frac{8}{3}R1_{ury} & T1_{ury} = 105 \\
T2_{urx} = 220 - \frac{2}{3}R1_{ury} & T2_{ury} = 110 \\
M_{x} = 400 - \frac{8}{3}R1_{ury} & M_{y} = 200 - R1_{ury} \\
R1_{urx} = \frac{4}{3}R1_{ury} & \\
 & \\
110 \leq R1_{ury} & R1_{ury} \leq 150
\end{array}
$$

Figure 7: Paremetric Solved form in R_{ury} - 61 constraints

constraints the system is over-constrained – it only has one solution. Note that this was certainly not obvious in the original set of constraints in non-canonical form.

At this point we need to determine which anchor constraints need to be removed. To do this, a set of inequations are added corresponding to the variables in the object that we wish to manipulate. This of course results in an unsatisfiable system of constraints, but, in computing the new FCF, the unsatisfiable system can be analyzed to find other constraints, other than those we just added, which cause the unsatisfiability. These can be then removed by the user.

If we consider only two anchor constraints instead of three, as in Figure 3, the ACF, as shown in Figure 7, is in parametric solved form having parameter $R1_{ury}$. This solved form consists of 59 equations and 2 inequalities,

$$110 \leq R1_{ury} \qquad R1_{ury} \leq 150 \qquad (1)$$

and clearly has only one degree of freedom. This solved form can now be compiled into code to update the dependent variables whenever this parameter is changed.

If the ACF is not in solved form, then the constraint system is under-constrained. From the ACF it is straightforward to determine possible choices for additional anchor constraints. These are variables which are not uniquely determined by the desired parametric variables of the ACF.

Thus the architecture provides:

- Rapid re-satisfaction of existing constraints when a small number of parameters, such as location of a vertex, are changed.
- Detection of under-constrained system and identification of which parameters can be fixed to constrain it.
- Recognition of an over-constrained system and identification of anchor constraints responsible for the unsatisfiability.

When an object is selected for manipulation, the ACF is projected onto the object's variables giving the ranges of values that they may take while still satisfying the constraints. As the constraints are linear, the range will always be a convex polygon allowing it to be easily displayed. In the example, we saw that the upper right corner of $R1$ was constrained to move only on a diagonal line. In other situations it might be constrained to move inside a small region. In both cases it is possible to present these regions graphically to the user.

Thus the architecture provides:

- Computation of the range of values that a parameter can take and still leave the system satisfiable.

Interactive editors must support the definition and compilation of compound objects. Although this is straightforward in systems without constraints, the addition of constraints raises new issues. In particular, efficiently representing the constraints in the compound object, and determining which variables of a compound graphic object define the objects it contains.

An efficient representation for the constraints of a compound object can be obtained by computing the canonical form for all the constraints inside it.

Defining a compound object is more problematic. Intuitively, the user should be able to select a group of objects on the display, and then the editor should abstract the desired definition from this instance of the definition. Essentially this can be done as follows. Once the user has selected a set of objects to form the compound object, selected anchor constraints must then be removed to permit the compound object to be located freely, and finally a set of *definitional* variables which ensure that all its internal objects are well-defined – none of their variables are left unspecified – must be selected. The last step can be done by ensuring that the constraints in the compound object are parametric solved form for these definitional variables.

Projection can be used to simplify the constraints associated with the definition, acting as a type of compilation or partial evaluation. The idea is to project the constraints onto the definitional variables of the new object: this will remove local or internal variables from the constraints, and so simplify them.

For example suppose we were constructing a square compound object from the rectangle definition given earlier. The square object has the additional constraints that the x and y extents are equal. If we allow the square to be defined in terms of a diagonal and a point, then the square's local constraints are simpler than those of the rectangle.

Thus the architecture supports:

- The definition and compilation of compound objects.

5 Empirical results

We now present empirical results concerning the performance of this constraint technology. We consider the following operations which are typical of those which occur during an interactive session:

1. Testing the solvability of a set of constraints and computing their canonical form.
2. Adding new constraints or anchor points, and computing a new canonical form.
3. Generating a parametric solved form in terms of a set of specific variables that correspond to a point which is being dragged.

All these operations are handled by a new incremental constraint solving system we are developing. This system integrates algorithms for: testing solvability; computing the canonical form; performing Gauss-Jordan reduction to obtain parametric solved forms; and performing projections. This system is implemented in C++ on an IBM Risc System/6000, model 530 running AIX. Run times are measured in virtual CPU seconds.

In this evaluation, we use three sets of constraints as test data and time the operations described above. The results are shown in Figure 8.

The first test corresponds to the well known example of recursively nested quadrilaterals (see Figure 9). The initial set of 76 constraints consist of constraints that ensure that the endpoints of lines in the quadrilaterals touch, that the midpoints of the quadrilaterals form a parallelogram, and that the vertices of the embedded quadrilaterals are at the midpoints of the edges of the enclosing quadrilateral. It takes 0.06 seconds to transform this system into its canonical form which contains 64 constraints. (Remark: the twelve constraints that are eliminated are exactly those that constrain the midpoints

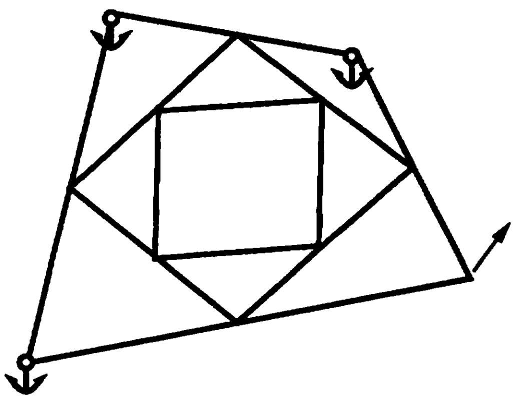


Figure 9: Recursively Nested Quadrilaterals

of each quadrilateral's edges to form a parallelogram. In retrospect, this was to be expected. The geometry theorem that this example illustrates states this fact!) To this system, we then add the three anchor points, consisting of 6 equality constraints, shown in Fig. 9. It takes 0.11 seconds to derive a new canonical form. Lastly, the time taken to obtain the parametric solved form in terms of the unanchored vertex of the quadrilateral is below the resolution of our timing system (1/100th of a second).

The second test corresponds to that given in section 3. The initial set of constraints (see Figure 5) consists of 82 constraints (54 equalities, 28 inequalities) over 60 variables. It takes 0.80 seconds to convert this to its canonical form (Fig. 6) containing 65 constraints. We then add 2 anchor points, consisting of 4 equalities. It takes 0.17 seconds to compute a new canonical form. Finally, the parametric solved form in terms of the point $(R1_{urx}, R1_{ury})$ takes 0.01 seconds to compute.

The third test, although from outside the graphics domain, has similar characteristics (sparsity, relative percentage of equalities and inequalities, etc..) as the previous examples. It is of interest because the initial constraints are all inequalities. The initial set of constraints consist of 1819 inequalities over 68 variables. This is simplified into 58 equalities plus 90 inequalities over 10 variables. Then, 10 new equalities are added and the set is updated accordingly. Finally, 10 variables are chosen to obtain a parameterized representation.

In these examples, the advantage of keeping constraints in canonical form is demonstrated by how little time it takes to add constraints, and to compute the parametric solved forms. The most time consuming operation is the detection of equalities implicitly defined by inequalities in the original system. This can be significant when there are large number of inequalities. However because the system is incremental, it will rarely, if ever, have to deal with the size and complexity of dataset 3 in an interactive application. Adding large numbers of inequalities at once

Total Constraints N (=,<)	Computing Canonical Form Constraints (=,<)	(sec)	Addition (sec)	Solved Form (sec)
Recursive Quads 76 (76,0) : 72 var.	64 (64,0)	0.06	0.11	<0.01
Layout: 82 (54,28) : 60 var.	65 (51,14)	0.80	0.17	0.01
Dataset 3: 1819 (0,1819) : 68 var.	148 (58,90)	17.96	0.28	0.01

Figure 8:

is unlikely.

6 Conclusion

We have shown how recent results and algorithms to manipulate linear arithmetic constraints provide a technology for interactive constraint-based user-interface systems. This extends previous constraint technology for user interfaces, by allowing simultaneous linear equations and inequalities, and by providing techniques which give improved user-feedback.

We have proposed an architecture based on this technology and are currently implementing a interactive constraint-based editor based upon it.

Acknowledgements

The authors thank Jean-Louis Lassez for his helpful comments.

References

[1] A. Borning, The programming language aspects of ThingLab – a constraint-oriented simulation laboratory, *ACM Trans. on Prog. Lang. and Systems* 3, 1981, 343-387.

[2] A. Borning, R.A. Duisberg. Constraint based tools for building user interfaces. *ACM Transactions on Graphics*. 5(4). 1986.

[3] D. Epstein and W.R. Lalonde, A Smalltalk window system based on constraints, in *Proc. of ACM Conference on Object Oriented Programming Systems, Languages and Applications (OOPSLA'88)*, pp. 83–94, ACM Press, 1988.

[4] R. Helm, K. Marriott and M. Odersky, Building visual language parsers, in *ACM CHI'91*, pp. 105–112, ACM Press, 1991.

[5] R. Helm and K. Marriott, Declarative specification of visual languages, in *1990 IEEE Workshop on Visual Languages*, pp. 98–103, IEEE Comp. Soc. Press, 1990.

[6] T. Huynh, C. Lassez and J-L. Lassez, *Fourier Algorithm Revisited*, 2nd International Conference on Algebraic and Logic Programming, Lecture Notes in Computer Sciences, Springer-Verlag 1990.

[7] T. Huynh, J-L. Lassez and K. McAloon, *Simplification and Elimination of Redundant Linear Arithmetic Constraints*, proc. NACLP 89, MIT Press.

[8] J-L. Lassez, *Querying Constraints*, Proceedings of the ACM conference on Principles of Database Systems Nashville 1990.

[9] C. Lassez and J-L. Lassez, *Quantifier Elimination for Conjunctions of Linear Constraints via a Convex hull Algorithm*, IBM Research Report. RC 16779. 1991.

[10] J-L. Lassez and K. McAloon, *A Canonical Form for Generalized Linear Constraints*, IBM Research Report RC 15004, IBM T.J. Watson Research Center, to appear Journal of Symbolic Computation.

[11] J.H. Maloney, A. Borning and B.N. Freeman-Benson, Constraint technology for user-interface construction in Thinglab II, *OOPSLA '89*, pp. 381–388, ACM Press, 1989.

[12] G. Nelson, Juno: a constraint-based graphics system, in *ACM SIGGRAPH' 85 Conf. Proc.*, pp. 235-243, ACM Press, 1985.

[13] D.R. Olson Jr., K. Allan, Creating Interactive Techniques by Symbolically Solving Geometric Constraints. *Proc. of Third Symp. on User Interface Software and Technology* Snowbird, Utah. October 1990. pp. 102-107.

[14] I.E. Sutherland, Sketchpad: A man-machine graphical communication system, in *Proc. of the Spring Joint Computer Conference*, pp. 329-345, 1963.

[15] P.A. Szekely and B.A. Myers, A user interface toolkit based on graphical objects and constraints, in *OOPSLA'88*, pp. 36–45, ACM Press, 1988.

[16] A. Witkin, M. Gleicher, W. Welch. Interactive Dynamics. in *Proc. of SIGGRAPH'90.* 1990

[17] C.J. Van Wyk, A high-level language for specifying pictures, *ACM Transactions on Graphics* 2, 1982, 163-182.

NON-UNIFORM PATCH LUMINANCE FOR GLOBAL ILLUMINATION

Buming Bian
University of Texas System — CHPC
Balcones Research Center
Commons Building, 1.154
Austin, TX 78712

Norman Wittels
Department of Civil EngineeringWorcester Polytechnic Institute
Worcester, MA 01609

Donald S. Fussell
Department of Computer Science
University of Texas at Austin
Austin Texas 78712

ABSTRACT

A new radiosity model is presented in which all patches are represented as isoparametric elements and the patch luminances change bilinearly. The surfaces are tessellated into planar quadrilaterals and continuous surface luminance is maintained where patches meet. A new form factor, accounting for the luminance contributions between patches, is derived and calculated using hemisphere projections and Gaussian quadrature. Images generated from the new approach were tested by pixel-level comparison with real images acquired by a calibrated imaging system, and compared with the images generated by the uniform patch luminance radiosity. The comparison results indicate that fewer bilinear patches are required to achieve comparable luminance accuracy.

CR Categories and Subject Descriptors: I.3.3 [Computer Graphics]: Picture/Image Generation; Display Algorithm; I.3.7 [Computer Graphics]: three-dimensional Graphics and Realism.

General Terms: Algorithms.

Additional Key Words and Phrase: radiosity, accurate image, non-uniform patch luminance, pixel level comparison, image registration.

1. INTRODUCTION

Radiosity methods for photo-realistic image generation have recently been the subject of intensive research interest in computer graphics. These methods involve the accurate determination of surface luminances before the pixel image is rendered. Since each diffuse surface receives light from and emits light to other surfaces visible to it, this interreflection problem can be modelled as a set of equations relating the luminance of each surface to the luminances of all other surfaces in the scene.

The interreflection problem is equivalent to the radiosity method as used in thermal engineering studies of radiative heat transfer[10]. Surface luminance is a function of the light from sources and the interreflected light from other diffuse surfaces. The key concepts are energy conservation (all light leaving a surface must be accounted for) and energy equilibrium (the total light out of any surface equals the total light in times the surface reflectivity). In the radiosity calculation, surfaces in a scene are divided into planar patches. The luminance at any point on a patch due to the light emitted by another patch can be calculated using the analytical methods summarized in [12, 14, 17]. The luminance can be expressed as:

$$L = \rho \int_{A_s} L_s \frac{\cos\theta_k \cos\theta_l}{\pi r^2} da_s \tag{1}$$

where ρ is the surface reflectance, L_s is the source luminance, r is the distance between the point and the source, θ_k is the angle between r and the normal of the point, θ_l is the angle between r and the normal of the source, and the da_s is the differential area on the source.

Current radiosity methods assume that the luminance is uniform on each patch[9]. Thus, if we use only average luminance values for each patch in a set of M patches, then L_k, the luminance of patch k, is a linear function of the luminances of the other patches, plus the self-illumination term L_k^0:

$$L_k = \rho_k \sum_{l=1}^{M} F_{kl} L_l + L_k^0 \tag{2}$$

where ρ_k is the reflectance of patch k, and the form factor F_{kl}, given by

$$F_{kl} = \frac{1}{A_k} \int_{A_k} \int_{A_l} \frac{cos\theta_k cos\theta_l}{\pi r^2} da_l da_k$$

is a function of the relative positions and orientations of the two patches. A_k is the area of patch k, da_l and da_k are, respectively, the differential area on patch l and k. For an enclosed scene, additional information can be obtained by using the following three principles. Energy conservation requires that the energy leaving a surface must equal the energy into that surface times the surface reflectivity. That is,

$$\sum_{l=1}^{M} \frac{A_k}{A_l} F_{kl} = 1 \tag{3}$$

for $k = 1, \ldots, M$, where A_k is the area of patch k. Reciprocity (the Second Law of Thermodynamics) requires that:

$$A_k F_{kl} = A_l F_{lk}$$

and the principle that no planar polygon may illuminate itself can be written as:

$$F_{kk} = 0$$

A number of algorithms [1, 2, 5, 6, 7, 8, 13, 16] have been developed for computing form factors and for solving the interreflection problem. All assume patches of uniform intensity. This assumption simplifies the calculation of form factors, but it can in many cases require that a scene be subdivided into a number of patches.

In this paper we develop a new radiosity model which allows the luminance of surface patches to vary bilinearly across their patches. We develop an effective approach to calculate form factors for such patches by using hemispherical projections and Gaussian quadrature in situations when analytic solutions do not exist. We then undertake a systematic comparison of images produced with bilinear luminance patches and uniform luminance patches to determine the effectiveness of our model in increasing accuracy with a smaller number of patches. This is done by comparing these images against a calibrated real would test scene.

The paper is organized as follows. Section 2 presents our radiosity model and develops a form factor function from a bilinearly illuminated quadrilateral patch to a receiver vertex. Section 3 shows how to compute such form factors using hemisphereical projection and Gaussian quadrature. Section 4 contains the experimental results of our accuracy tests.

2. NEW RADIOSITY APPROACH

The basic question addressed in this section is as follows: Given that a patch has non-uniform luminance, what mathematical model of that luminance will be sufficiently accurate, yet still allow for practical calculation of interreflection form factors? Any surface luminance distribution which is an analytic function $f(x, y)$ of the coordinates has a Taylor's series expansion:

$$f(x,y) = f(x_1,y_1)+f_x(x_1,y_1)(x-x_1)+f_y(x_1,y_1)(y-y_1)$$
$$+f_{xy}(x_1,y_1)(x-x_1)(y-y_1)+\cdots$$

where $f(x_1, y_1)$, $f_x(x_1, y_1)$, and $f_y(x_1, y_1)$ are, respectively, the value and partial derivatives of the function at the point (x_1, y_1) and $f_{xy}(x_1, y_1)$ is the mixed second derivative. Standard radiosity approaches assume that the luminance is constant, therefore they use only the zero order term $f(x_1, y_1)$ to describe the surface luminance of a patch, that is, $f(x, y) = f(x_1, y_1)$ for a given polygonal patch. To model non-uniform luminances, at least the first order or linear terms need be used. A continuous luminance model and quadrilateral tessellation model based upon the crossterm Taylor series expansion is used in our discussion.

The interior luminance, in a quadrilateral patch, may be expressed as a bilinear interpolation of the luminances at the four vertices $L_j (j = 0, 1, 2, 3)$:

$$L = \sum_{j=0}^{3} N_j L_j \tag{4}$$

where the N_j, called shape functions, depend only upon the shape of the quadrilateral. Their explicit forms are given as follows:

$$N_0 = \frac{1}{4}(1-s)(1-t) \quad N_1 = \frac{1}{4}(1+s)(1-t)$$

$$N_2 = \frac{1}{4}(1+s)(1+t) \quad N_3 = \frac{1}{4}(1-s)(1+t)$$

where s and t are parameters, satisfying $|s| \leq 1$, $|t| \leq 1$. Shape functions satisfy the following rules: 1) The mapping between the a plane in the xyz space and the st space is homeomorphic. 2) Each shape function has value one at its own vertex and zero at others. 3) Each shape function is zero along any edge that does not contain its vertex. 4) Each shape function is a polynomial of the same degree as the interpolation equation.

In the simulation, triangles and quadrilaterals were selected as primitives. It is easy to show that the triangle is basic; any planar polygon can be exactly decomposed into triangles. Since triangles are degenerated quadrilaterals with two vertices coinciding, all the patches will be represented as quadrilaterals.

Now consider the interreflection between two diffuse patches with luminance described by Eq.(1). From Eq.(4) , the luminance contribution of patch l to the ith vertex of patch k, (see Fig.1), is:

$$L_i = \rho_k \int_{A_l} \sum_{j=0}^{3} N_j L_j \frac{cos\theta_i cos\theta_j}{\pi r^2} da_l = \rho_k \sum_{j=0}^{3} K_{ij} L_j$$

where ρ_k is the diffuse reflectivity of patch k, L_j is the luminance of the jth vertex of patch l, da_l is a differential area on patch l, and

$$K_{ij} = \int_{A_l} N_j \frac{\cos\theta_i \cos\theta_j}{\pi r^2} da_l \tag{5}$$

is the new form factor corresponding to the bilinear patch luminance distribution.

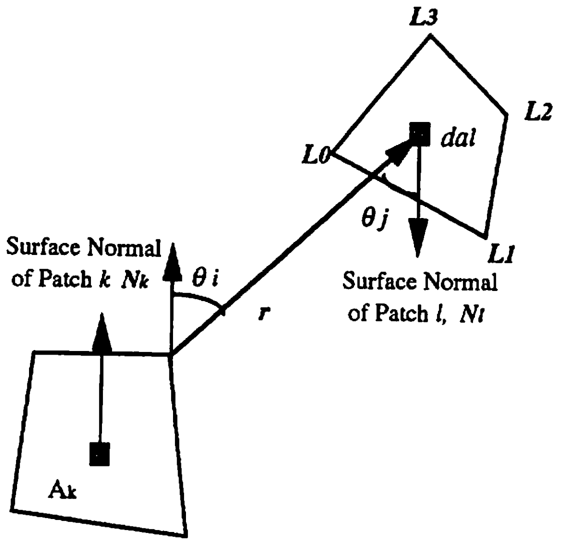


Fig.1 Luminance contribution from one patch to one vertex of another patch

Therefore, the total luminance from patch l to the vertex i of patch k is a linear combination of the vertex luminances of patch l, and vice versa. For an enclosed scene with M patches, there will be $4M$ vertices. With self-luminance included, the luminance on each vertex can be written as

$$L_i = \rho_k \sum_{j=1}^{N} K_{ij} L_j + L_i^0 \tag{6}$$

where $N \leq 4M$. $4M$ is an upper bound; the actual number may be lower. Many of the vertices coincide where patches meet and the redundancies can be eliminated. For a large surface subdivided into K^2 patches, there are only $(K+1)^2$ independent vertices. In Eq.(6), i is the vertex index of patch k and j is the vertex index of patch l, with $k = (i \ mod\ 4)$ and $l = (j \ mod\ 4)$.

Before calculating the form factors, the physical correctness of the new model needs to be verified. That is, we need to see if this model follows the energy conservation and reciprocity laws. To maintain energy conservation, the energy leaving a surface must equal the energy arriving at this surface times the reflectivity. The luminance at a point on patch k equals the summation of all the contributions from the whole scene and the total flux into the patch is

$$\begin{aligned}\Phi_{in} &= \frac{\pi}{\rho_k} \int_{A_k} L_k da_k \\ &= \frac{\pi}{\rho_k} \int_{A_k} \sum_{i=0}^{3} N_i L_i da_k \\ &= \frac{\pi}{\rho_k} \sum_{i=0}^{3} L_i A_i\end{aligned}$$

From Eq.(6), the total flux out of the patch k is the summation of the luminances received by the rest of the patches:

$$\begin{aligned}\Phi_{out} &= \sum_{l=1}^{N} \pi \int_{A_l} L_l da_l \\ &= \pi \sum_{l=1}^{N} \int_{A_l} \sum_{i=0}^{3} K_{ji} L_i da_l \\ &= \pi \sum_{i=0}^{3} L_i B_i\end{aligned}$$

where:

$$A_i = \int_{A_k} N_i da_k$$

$$B_i = \sum_{l=0}^{N} \int_{A_k} \int_{A_l} N_i \frac{cos\theta_i cos\theta_j}{\pi r^2} da_l da_k$$

Energy conservation requires that:

$$\rho_k \Phi_{in} = \Phi_{out}$$

that is

$$\sum_{i=0}^{3} A_i = \sum_{i=0}^{3} B_i$$

Consider a special case when only the jth vertex has non-zero luminance. In that case,

$$A_j = B_j \qquad \text{for } j = 0, 1, 2, 3.$$

A_i is the area integral of shape function N_i, because:

$$\sum_{i=0}^{3} A_i = \text{Area of patch } k$$

This can be written in a form comparable to Eq.(3):

$$\frac{1}{A_i} \sum_{l=0}^{N} F_{il} = 1$$

where

$$F_{il} = \int_{A_k} \int_{A_l} N_i \frac{cos\theta_i cos\theta_j}{\pi r^2} da_l da_k$$

Reciprocity can be written in a form comparable to :

$$F_{il} = F_{jk}$$

where i and j satisfy the condition: $i \ mod\ 4 = j \ mod\ 4$. By virtue of this constraint, the shape functions involved in the form factors are the same. Therefore, it is guaranteed that the F_{il} are the same. The physical meaning of this equation is that the fraction of the flux leaving patch k and arriving at patch l equals to the fraction of the flux leaving patch l and arriving at patch k, therefore the system is stable.

3. CALCULATION OF THE NEW FORM FACTOR

In this section, hemisphere projection[3, 4] is used for calculating the form factor of Eq.(5) for an arbitrary convex planar polygon. See Fig.(2) for an example of the hemispherical projection of a triangular patch. The edges of any diffusely-emitting planar polygon which illuminates the point at the center of the hemisphere will intersect the sphere in arcs which are portions of great circles. Those arcs orthogonally project into portions of great ellipses on the equatorial plane of the hemisphere, these portions of great ellipses form an elliptical polygon on the equatorial plane. The area of that elliptical polygon is proportional to the illuminance of the original planar polygon. What we present here is the basis for an algorithm to calculate the illuminance at an arbitrary point P due to the light emitted by a luminous planar triangular patch.

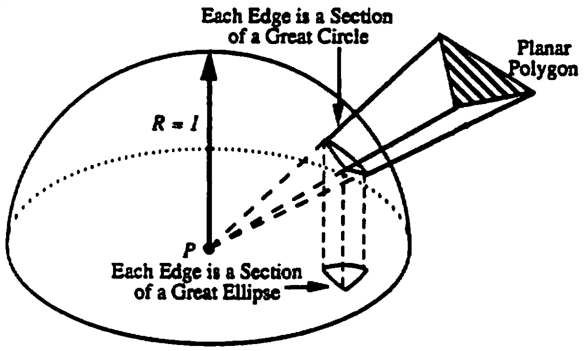

Each Edge is a Section of a Great Circle
Planar Polygon
R = 1
P
Each Edge is a Section of a Great Ellipse

Fig.2 Hemispherical projection of a triangular patch

1) Construct a unit hemisphere with its center at the point P and its equatorial plane tangent to the surface containing P.

2) Project the three vertices of the planar triangle on to the unit hemisphere using lines that pass through the center of the sphere. Given three points: A, B, and C, we have three unit vectors **A**, **B**, and **C** as following:

$$\mathbf{A} = \frac{\overrightarrow{PA}}{|\overrightarrow{PA}|} \qquad \mathbf{B} = \frac{\overrightarrow{PB}}{|\overrightarrow{PB}|} \qquad \mathbf{C} = \frac{\overrightarrow{PC}}{|\overrightarrow{PC}|}$$

3) Every pair of points lies on a uniquely determined great circle; the intersection of the three great circles form a spherical triangle, which is the spherical projection of the planar triangle. Arbitrarily select two vectors **A** and **B**. The normal to the great circle containing **A** and **B** is:

$$\mathbf{N} = -k_1(\mathbf{A} \times \mathbf{B})$$

where k_1 is the normalization factor for **N**.

4) The spherical triangle is projected on to the equatorial plane forming a planar elliptical triangle whose edges are arcs of uniquely determined great ellipses. The major axis vector $\mathbf{M}_1$ and minor axis vector $\mathbf{M}_2$ are:

$$\mathbf{M}_1 = k_2(\mathbf{Z} \times \mathbf{N})$$

$$\mathbf{M}_2 = \mathbf{Z} \times \mathbf{M}_1 = k_2(\mathbf{Z} \times (\mathbf{Z} \times \mathbf{N}))$$

where k_2 is the normalization factor for $\mathbf{M}_1$, and note that $|\mathbf{M}_1| = 1$, $|\mathbf{M}_2| \leq 1$.

5) Using the standard formula for the area of an elliptical sector, the area of the elliptical triangle is computed. This is proportional to the illuminance produced by the planar triangle.

$$F_i = \frac{|\mathbf{M}_2|}{2}\left(a\cos\left(\frac{\mathbf{B}\bullet\mathbf{M}_2}{\mathbf{B}\bullet\mathbf{M}_1}\right) - a\cos\left(\frac{\mathbf{A}\bullet\mathbf{M}_2}{\mathbf{A}\bullet\mathbf{M}_1}\right)\right) \tag{7}$$

And the form factor is now:

$$F = \sum_{i=1}^{3} F_i$$

Gaussian quadrature can be applied to the 2-D form factor integral of Eq.(5) when the hemisphere projection has no analytic solution. The Gaussian quadrature method multiplies the function evaluated at a matrix of interpolation points with known weights for each point. The integral is approximated as a summation of these weighted values:

$$\int f(s,t)dsdt = \sum_{i=1}^{N_1}\sum_{j=1}^{N_2} f(s_i,t_j)W_iW_j \tag{8}$$

The accuracy of the approximation depends upon the number of interpolation points, N_1 and N_2. The differential area of a patch can be expressed as a vector which is the cross product of two vectors lying on the patch

$$d\mathbf{a} = d\mathbf{l}_s \times d\mathbf{l}_t$$

where

$$d\mathbf{l}_s = (x_s\mathbf{x} + y_s\mathbf{y} + z_s\mathbf{z})ds$$
$$d\mathbf{l}_t = (x_t\mathbf{x} + y_t\mathbf{y} + z_t\mathbf{z})dt$$

with **x**, **y**, and **z** are the unit vectors along x, y, and z axis, x_s and x_t are the partial derivatives of x with respect to s and t, similarly for y_s, y_t, z_s, and z_t. Then with the interpolation method, an arbitrary 3-D planar quadrilateral in xyz space can be transformed to a 2-D square in the st space[11], (see Fig.3), with sides of length of 2:

$$x = \sum_{j=0}^{3} N_j x_j \qquad y = \sum_{j=0}^{3} N_j y_j \qquad z = \sum_{j=0}^{3} N_j z_j.$$

With $d\sigma = dsdt$, the directional differential can be expressed as:

$$d\mathbf{a} = (S_x\mathbf{x} + S_y\mathbf{y} + S_z\mathbf{z})d\sigma$$

The form factor K_{ij} can then be computed by Gaussian quadrature:

$$K_{ij} = \int_{\Re_{st}} N_j \frac{(\mathbf{N}_k \bullet \mathbf{r})(\mathbf{N}_l \bullet \mathbf{r})}{\pi r^4} det|J|d\sigma$$

The form factor K_{ij} can then be computed by Gaussian quadrature:

$$K_{ij} = \int_{\Re_{st}} N_j \frac{(\mathbf{N}_k \bullet \mathbf{r})(\mathbf{N}_l \bullet \mathbf{r})}{\pi r^4} det|J| d\sigma$$

where $\mathbf{r}$ is the vector from vertex i on patch k to the differential area da_l, $det|J|$ is the determinant of the Jacobian transformation matrix from xyz space to st space. Note here that J is only a 2×2 matrix; since the patch is planar there are only two independent variables among x, y, and z. It is a quantity which includes the weight due to shape functions and the space transformation. The sign of $(\mathbf{N}_k \bullet \mathbf{r})$ and $(\mathbf{N}_l \bullet \mathbf{r})$ can be used to decide whether or not the illuminating patch k is facing the patch l. By now, the integral has been transformed to the st domain over a 2×2 square, and Eq.(8) can be applied directly for form factor calculation.

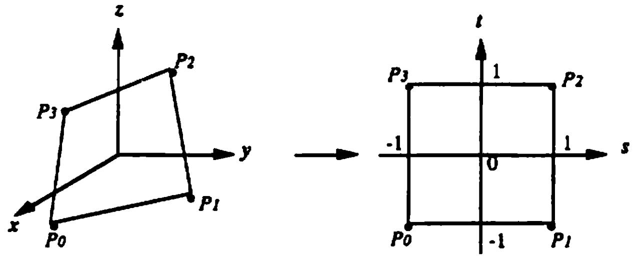
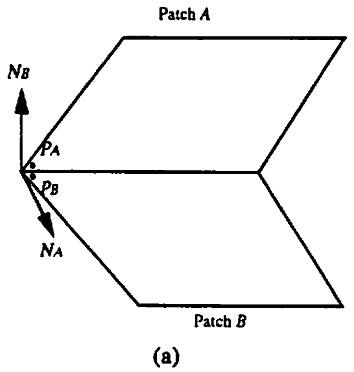
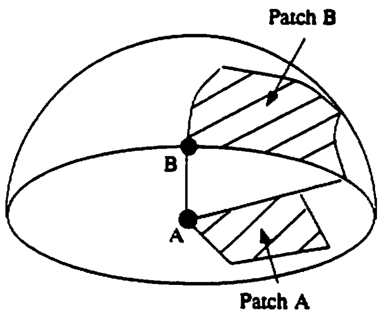


Fig.3 Isoparametric transformation from xyz space to st space

It is useful to describe a situation in which hemispherical projection leads to an analytic solution where numerical solution would not converge. This is the problem raised by considering the luminance on the vertices instead of the center of the patch. Consider the case in which two planar patches, not lying in the same plane, meet along a common edge, i.e., the point P_A on patch A coincides with point P_B on patch B. The luminance at the point P_A due to the patch B will involving all the points on the patch B, including P_B. Since the distance between P_A and P_B is zero, Gaussian quadrature will not give correct result. While in the hemisphere projection, P_A is at the center of the hemisphere, and the patch P_B will be projected on to the hemisphere. Then the form factor can be calculated analytically by the process described in [4].



(a)



(b)

Fig.4 Contribution from close points on a neighboring patch

4. SIMULATIONS, EXPERIMENTS, AND COMPARISONS

Visual assessment, the primary means for judging the realism of images, is inadequate for assessing simulation accuracy because the human eye is not capable of absolute luminance measurement. Our experiments provide a more objective accuracy comparison in the following way. First, images were simulated using the new radiosity model. Then corresponding real scene images were obtained and objective methods for comparing them with simulations were developed. Lastly, the images simulated by uniform luminance and non-uniform luminance assumptions ware compared to give a validated measure of the new method.

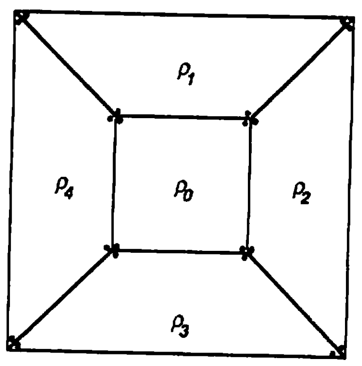


Fig.5 Vertices definition on the surfaces of a box

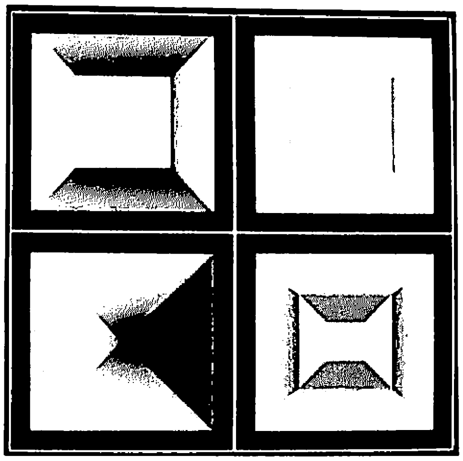


Fig.6 Images generated by new radiosity method

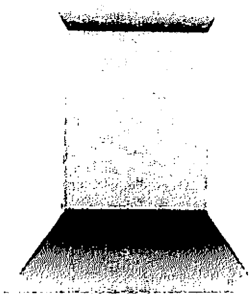


Fig.7 Simulated image of a box

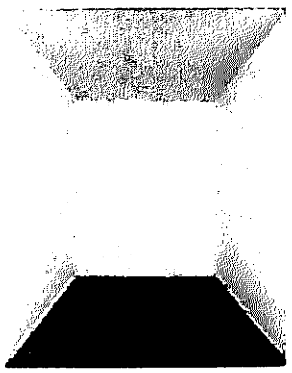


Fig.8 Real image of the box corresponding to Fig.7

To test the new radiosity model, simulations of images of real objects is necessary. Scenes were simulated out of environments made of boxes and pyramids. The reason for using such simple figures is that we wanted to be able to construct physical replicas that could be compared with synthesized images. The luminances on the internal vertices of patches in the synthesized images were calculated to simulate the read scene luminances. Fig.5 shows the twenty four internal vertices of a box made of materials with diffuse reflectivities ρ_0 through ρ_5. Note that the front face has been removed and displayed to the right to allow looking inside the box. The simulated images are shown in Fig.6, upper left is a box, upper right is a box with two corners cut, lower left is the cube connected to a pyramid, and lower right is the cube connected to a pyramid with the top of the pyramid cut.

To quantitatively evaluate the simulated images, a simple experimental model (the inside of a box fabricated with diffuse cardboard with various reflectivities) was constructed and photographed. It is impractical to photograph the inside surfaces of a closed box so we used an analogous arrangement. Lights were carefully arranged to produce a uniform illuminance on a diffuse translucent surface with a hole in it. A camera looked through the hole into the open side of the box. All of the box surfaces have different reflectivities. The dimensions and measured reflectivities of the box materials were used in generating the simulated image in Fig.7. The corresponding real digital image produced by this experiment is shown in Fig.8.

Before images can be compared at the pixel level, they must be registered to compensate for the unavoidable differences between the real and simulated imaging conditions. That means that the image gray levels must be normalized and affine transformations must be applied to account for translation and rotation of objects and for perspective errors caused by incorrect lens focal length or camera orientation.

It may be necessary to apply some sort of low-pass filtering, such as Gaussian filtering, to the images to prevent aliasing effects. This is often done when registering images[15]. The generated image was resampled to match the size of the real image to the sub-pixel range and a Mean Square Error Root was computed:

$$MSE(m,n) = \sqrt{\frac{\sum_{i=0}^{N_1}\sum_{j=0}^{N_2}[f_1(i+m,j+n) - \gamma f_2(i,j)]^2}{N_1 N_2}}$$

where m and n were the distances the simulated image was moved in the 2-D pixel space, f_1 and f_2 are the pixel luminances of the simulated and real images, N_1 and N_2 are image height and width in pixels. Homogeneous transformations were applied to one of the images and the gray level scale factor γ was adjusted until the MSE was minimized. This residual MSE is composed of three components: noise and other errors in the image capture system; round-off and other errors that are inherent in the resampling process; and, simulation inaccuracy. By measurements on the image capture system and by calculations and studies of resampled test images, the first two sources of error have been quantified. The remaining error is the simulation accuracy error. This was computed for the real and simulated images of boxes.

The big luminance difference on the side faces between the real and simulated images can be qualitatively explained as following. The major reason is that in the case of simulation, the distance from the source to the receptor ranges from 0 to 2, while for the real image, the range is from 2 to 4 since the box is open for the light source and the camera. The corresponding luminance ratio by [14] is:

$$\frac{I(0) - I(2)}{I(2) - I(4)} = 36.55$$

Due to the interreflection, the actual ratio will be smaller than this number, but the effect will be very significant. The back face of the real image is not affected much. Then normalization is applied, and it raises the overall luminance of the real image. That is why the real image has much lower contrast than the simulated one.

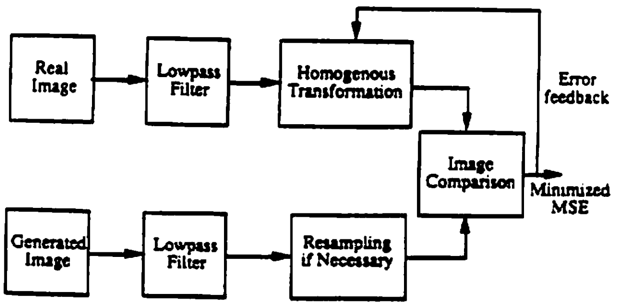


Fig.9 Image registration process

The comparison process is shown in Fig.9. Resampling registers the sizes of the real and simulated images for MSE comparison. Homogeneous transformation places the two images in corresponding positions and orientations. Modification is necessary because the simulation modeled a closed environment but the experimental environment is an open one. Using hemisphere projection techniques this error can be calculated and compensation can be applied, producing a modified simulated image (see Fig.10) with residual MSE of 4.2%. An error image of the real and modified simulation shows that most of the remaining error is due to misalignments along the edges between box surfaces. Some of that error is due to the difficulties in constructing a box to sub-pixel accuracy, so the measured MSE is probably an upper bound. The actual edge locations could be measured and used as inputs to the simulation if a better estimate is required. Though the comparison methods could be refined further, the current results show that the new radiosity method produces accurate simulations with accuracy near the noise limit of the image capture system.

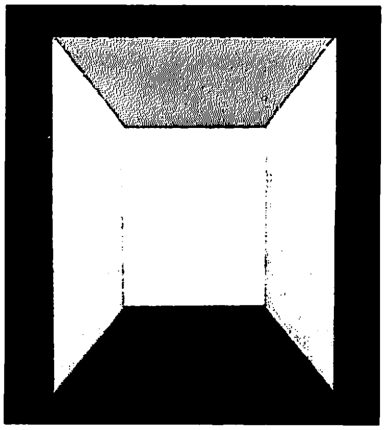
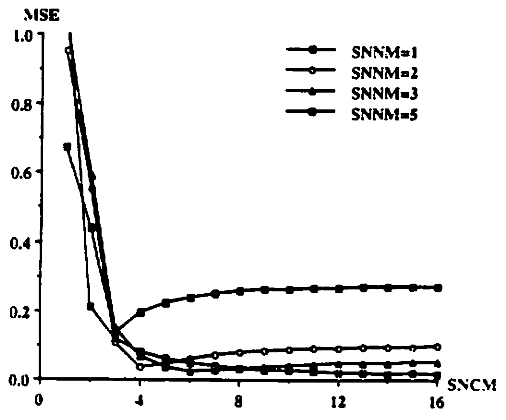


Fig.10 Modified simulation image



(a)

SNNM	1	2	3	4	5	6
SNCM	3	4	6	8	12	16

(b) Best Match List:

Fig.11 Patch number comparison between new and current radiosity methods

Fig.11 shows that the new radiosity method requires fewer patches to produce same accuracy as current radiosity methods. In Fig.11, SNCM means Subdivision Number of Current Method, with each surface divided into SNCM^2 patches; SNNM means Subdivision Number of New Method with each surface divided into SNNM^2 patches. Fig.11(a) shows the MSE between the images by new method and those by current methods, Fig.11(b) shows the best match list between SNCM and SNNM. In the worst case, the new approach uses one fourth of number of the patches required by current radiosity methods.

5. CONCLUSIONS

A new radiosity model with non-uniform and continuous patch luminance distribution has been presented. Hemispheric projection was used to calculate the form factor analytically where possible. Otherwise Gaussian quadrature is applied. Images for simple 3-D closed scenes were simulated with the new approach. The corresponding real image from a calibrated imaging system was compared, on a pixel-to-pixel basis, instead of the human visual assessment. MSE measurements were used to evaluate the simulated image and the comparison results are presented which demonstrate that very accurate simulations of a simple environment can be achieved using the model. Images generated by uniform patch luminance radiosity method were compared with the images produced by the non-uniform method, to determine the increase in the number of patches required to achieve a given accuracy using uniform luminance instead of non-uniform luminance, the new radiosity model was proved.

The new radiosity model allows greater flexibility in tessellation and provides accurate interreflection calculations with fewer quadrilateral patches. The quadrilateral tessellation is desirable when curved surfaces are used, tessellation into Bezier patches is most easily accomplished using quadrilateral patches. Also, the new luminance model naturally uses quadrilaterals because of the coordinate transformations that are performed.

Since the luminance distribution is done as part of the interreflection calculation, the interreflection and shading calculations are performed simultaneously. Therefore the consistency of scene luminance simulation and digital image rendering has been kept. This alleviates the need to use an ad hoc interpolation scheme during rendering. Although substantially reduced when compared to previous radiosity methods, Mach band effects remain at common edges where patches meet. This is due to the fact that although the luminances are continuous, the directional derivatives may not be. It appears to therefore be necessary to use higher order approximation to the surface luminance to totally eliminate Mach band effects.

ACKNOLEDGEMENTS

We would like to acknowledge the support of National Science Foundation under contract No.87008872. Most of this work was performed in the Department of the Electrical Engineering at Worcester Polytechnic Institute. Thanks goes to Gongxian Liu for her help in typing manuscript, to Dr. Matthew Witten for his helpful suggestions and encouragement, to Dr. James Almond for his support.

REFERENCES

[1] Daniel R. Baum, Stephen Mann, Kevin P. Smith, and James M. Widget. *"Making Radiosity Usable: Automatic Preprocessing and Meshing Techniques for the Generation of Accurate Radiosity Solutions,"* Computer Graphics, *(Proc. SIGGRAPH'91), Vol. 25, No. 4, pp51–60, 1991.*

[2] Daniel R. Baum, Holly E. Rushmeier, and James M. Widget. *"Improving Radiosity Solutions Through the Use of Analytically Determined Form-Factors,"* Computer Graphics, *(Proc. SIGGRAPH'89), Vol. 23, No. 3, pp.325–334, 1989.*

[3] Buming Bian. Accurate Simulation of Scence Luminance, *Ph. D. Thesis, Worcester Polytechnic Institute, Worcester, MA., June, 1990.*

[4] Buming Bian and Norman Wittels. *"Accurate Image Simulation by Hemisphere Projection,"* Proceedings of SPIE/IS&T, *Vol. 1453, San Jose, CA, February, 1991.*

[5] A.T. Campbell, III and Donald S. Fussell. *"Adaptive Mesh Generation for Global Diffuse Illumination,"* Computer Graphics, *(Proc. SIGGRAPH'90), Vol. 24, No. 4, pp.155–164, 1990.*

[6] Shenchang Eric Chen. *"Incremental Radiosity: An Extension of Progressive Radiosity to an Interactive Image Synthesis System,"* Computer Graphics, *(Proc. SIGGRAPH'90), Vol. 24, No. 4, pp.135–144, 1990.*

[7] Shengchang Eric Chen, Holly E. Rushmeier, Gavin Miller, and Douglass Turner. *" A Progressive Multi-Pass Method for Global Illumination,"* Computer Graphics, *(Proc. SIGGRAPH'91), Vol. 25, No. 4, pp.165–174, 1991.*

[8] Michael F. Cohen, Shenchang Eric Chen, John R. Wallace, and Donald P. Greenberg. *"A Progressive Refinement Approach to Fast Radiosity Image Generation,"* Computer Graphics, *Vol.22, pp.75-84, 1988.*

[9] Cindy M. Goral, Kenneth E. Torrance, Donald P. Greenberg, and Bennett Battaile. *"Modeling the Interaction of Light Diffuse Surfaces,"* Computer Graphics, *Vol.18, pp.213-222, 1984.*

[10] H.C. Hottel and A.F. Sarofim. Radiative Transfer, *McGraw-Hill, 1967.*

[11] Thomas J.R. Hughes. The Finite Element Method: Linear Static and Dynamic Finite Element Analysis, *Prentice-Hall, Inc., 1987.*

[12] Philip Cooper Magnusson. Development of Methods of Solving Linear Inhomogeneous Equations of the Second Kind Arising in Interreflection and Other Electrical Engineering Field Problems, *Ph. D. Thesis, MIT, 1941.*

[13] D.S. Immel Michael F. Cohen, Donald P. Greenberg and P.J. Brock. *"An Efficient Radiosity Approach for*

[14] Parry Moon. The Scientific Basis of Illuminating Engineering, *McGraw-Hill, 1936.*

[15] Harold J. Reitsema, Allen J. Mord, and Eric Ramberg. *"High-Fidelity Image Resampling for Remote Sensing,"* Proc. SPIE, *Vol.432, pp.211-215, 1983.*

[16] John J. Wallace, Micheal F. Cohen, and Donald P. Greenberg. *"A Two Pass Solution to the Rendering Equation: A Synthesis for Ray Tracing and Radiosity Methods,"* Computer Graphics, *Vol.21, pp.31-40, 1987.*

[17] Ziro Yamanouti. *"Geometrical Calculation of Illumination due to Light from Luminance Sources of Simple Forms,"* Res. Electrotech. Lab., *Tokyo, Vol.148, October, 1924.*

A Two-Pass Physics-Based Global Lighting Model

Kadi Bouatouch
Pierre Tellier*

IRISA, Campus de Beaulieu
35042 Rennes Cedex, FRANCE
Tel: +33 99.84.72.58, Fax: +33 99.38.38.32
kadi@irisa.fr, tellier@irisa.fr

Abstract

This paper describes a two-pass implementation of a physics-based global lighting model. This latter uses a physics-based reflection model, spectral distribution of light powers, and does not make any assumption on the specular behavior of materials. The scenes are discretized into points instead of patches. Hence, any kind of surface can be used without having to break down it into small planar patches. A data structure, named visibility graph, is built to efficiently evaluate the visibility between the sample points of the scene. Even though the photometric properties of surfaces (reflection, transmission, roughness, emitted powers...) are modified, this graph does not change, which makes it easy to produce very quickly several images. Methods for computing the Fresnel factor are given in appendix.

Résumé

Cet article décrit une mise en œuvre, en deux phases, d'un modèle d'éclairement global dérivé de la physique. Ce modèle utilise un modèle physique de réflexion, des densités spectrales d'énergie et ne fait aucune hypothèse sur l'aspect spéculaire des matériaux utilisés. Les scènes sont discrétisées en points et non en carreaux, ce qui permet d'utiliser n'importe quel type de surface, sans avoir à les subdiviser en petits carreaux planaires. Une structure de données, appelée graphe de visibilité, est introduite afin de déterminer efficacement la visibilité entre les points échantillons de la scène. Ce graphe reste invariant lorsque les propriétés photométriques des surfaces (réflexion, transmission, rugosité, énergies émises par les sources) sont modifiées, ce qui permet de produire rapidement plusieurs images d'aspect différent. Enfin, des méthodes de calcul du facteur de Fresnel sont données en annexe.

*CSTB, Eclairage et Colorimétrie, 11 rue Henri-Picherit, 44300 NANTES Cedex 03, FRANCE. Tel: +33 40 37 20 00, Fax: +33 40 37 20 40

1 Introduction

Photosimulation consists in producing highly realistic images. The realistic aspect of materials can be simulated only with the use of physics-based reflection and transmission models. Such models have been introduced in [11, 17, 22]. To accurately evaluate the illumination of synthesized scenes, a global model is required. The implementation of this global model can be performed according to three approaches: one-pass methods [24, 21, 25, 19], two-pass methods [28, 30], or multi-pass methods [26, 8]. The one-pass methods perform all the illumination computations independently of the view point, allowing then a fast rendering of the same scene from different view points. However, these methods need a large amount of memory to store data. Another drawback is the aliasing defects due to sharp variations of specular reflections and specular transmissions. To avoid these defects, a very dense sampling of the scene is indispensable, which would significantly increase the data to be stored.

In the two-pass methods, the diffuse and specular components (from reflection or transmission) are computed separately; the notion of form factors are then extended to account for the specular effects contributing to the global diffuse component. In our opinion, these methods seem efficient since they offer a good realism and a non prohibitive computing time.

Even though the multi-pass methods are better suited to the rendering of caustic effects, they are very time consuming since they involve several passes: Monte Carlo path tracing, light tracing, progressive refinement radiosity...

For the reasons quoted above, the global model described in this paper has been implemented according to a two-pass method. It uses a physics-based reflection model as well as a transmission model. In this model, the light powers emitted, reflected or refracted are represented by their spectral distribution, the materials are characterized by their spectral reflectance (Fresnel factor) and their spectral transmittance as well as their microscopic roughness.

As suggested in [20], all the light powers are sampled at four wavelengths. The used reflection model is Cook's and Torrance's model [11]. Moreover, our global model does not make any assumption on the specular behavior of materials.

As pointed out hereafter, in our method, the scene is discretized into points instead of small patches, which avoids the breaking down of all surfaces into small patches. So, any kind of object can be used.

This paper addresses the following subjects. First, the global lighting model is presented as well as the reflection and transmission models. Then, we describe the different processings involved by our implementation: discretization of the scene, expression of the discretized light energy balance equation, and the two passes. In section 4, a data structure, named ***visibility graph***, is described in details. It will be shown that this data structure reduces drastically the amount of time needed for computing the visibility between the sample points of the scene. Finally, some experimental results are given, and a comparison with other methods is made. Methods for computing the Fresnel factor are given in appendix.

2 The Global Model

2.1 System of Light Energy Balance Equations

A global illumination model must take into account all the reflections and refractions within the scene. To describe the mechanism of light transport we use the model introduced in [5, 6]. This model consists of a set of equations which express (in terms of *radiance* since it is the quantity the eye is sensitive to) the radiance of a point P_i in the direction of P_j when illuminated by all the surfaces S_k (see figure 1):

$$L_\lambda(P_i, P_j) = h(P_i, P_j) * \tag{1}$$

$$\left[L_\lambda^E(P_i) + \sum_k \int_{S_k} R_\lambda(P_k, P_i, P_j) L_\lambda(P_k, P_i) G(P_k, P_i) dS_k\right]$$

where

- $L_\lambda(P_i, P_j)$ is the radiance of P_i as seen from P_j (emitted light power per unit surface and unit projected solid angle),
- $h(P_i, P_j)$ is the visibility function,
- $L_\lambda^E(P_i)$ is the self-emitted radiance,
- $R_\lambda(P_k, P_i, P_j) = s_i R_\lambda^s(P_k, P_i, P_j) + d_i R_\lambda^d(P_i)$ is either the bidirectional reflectance [11], or the bidirectional transmittance [23],
- $L_\lambda(P_k, P_i)$ is the radiance of P_k as seen from P_i,
- $G(P_k, P_i) = \frac{\cos\alpha_i \cos\beta_k}{\|P_k P_i\|^2}$ is a purely geometrical term,
- λ is a given wavelength.

The system made up of these equations is called ***system of light energy balance equations***. Solving this system will provide the global radiance at each point of the scene.

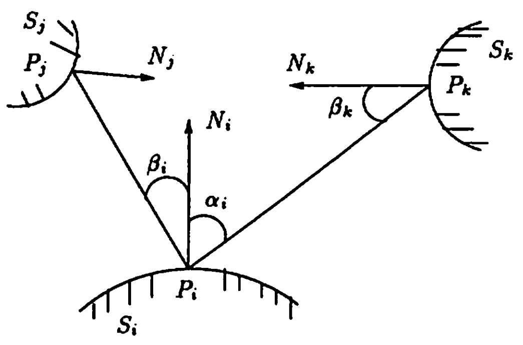


Figure 1: Geometry of light transport mechanism

2.2 Reflection Model

The used reflection model is the one proposed by Cook and Torrance [11]. With this model, the reflected light depends on the wavelength, the incidence angle, the roughness parameter, and the surface refractive index (this index is a complex number for metallic materials). This model takes into account the polarization of the light, the roughness and the masking/shadowing of the materials. Let us briefly review this model.

This model is expressed as:

$$R = sR_s + dR_d \quad \text{with} \quad s + d = 1$$

where R_d and R_s are respectively the diffuse and specular components, d and s are the proportions of the incident light which give rise to the diffuse and specular components respectively.

R_d is independent of the incident angle, and can be approximated by $\frac{F(\lambda,0)}{\pi}$ [11], where $F(\lambda, 0)$ is the Fresnel factor for a normal incidence.

R_s accounts for the roughness as well as for the masking/shadowing effects, and is expressed as:

$$R_s = \frac{1}{4\pi} \frac{F(\lambda,\theta).D.G}{\cos\theta_i \cos\theta_r},$$

where $F(\lambda, \theta)$ is the Fresnel factor, θ_i is the incidence angle (direction D_i), θ_r the reflection angle (direction D_r) and θ equals half of the angle $(D_i \hat{,} D_r)$. G is the masking/shadowing function, and D models the roughness effect. In our implementation, D is the Beckman function. The Fresnel factor is given by the Fresnel formula. In appendix, we show how this factor can be computed efficiently (even in case of metallic materials).

2.3 Transmission Model

So far, no physics-based transmission models have been proposed in the literature, but only an empirical one [14]. Rather than using an empirical transmission model, it is more realistic, for each material, to use transmittance values experimentally obtained with the help of a spectrophotometer [12].

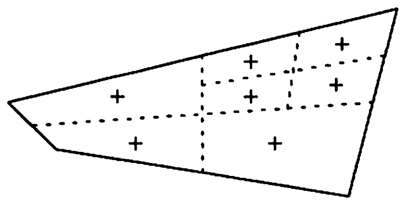


Figure 2: Sampling a polygon into points

However, if any transmission model exists, it can be easily integrated in our implementation. For this reason, at the present time, only ideal specular refraction is available in our method. In case of ideal specular refraction, R_s is no more than $1 - F(\lambda, \theta)$, and $s = 1$.

3 The Method

3.1 Discretizing the Scene Into Points

3.1.1 Motivation

Any method used to implement a global illumination model requires a discretization of the scene either into patches or points. In our method, we have chosen to discretize the scene into points for the following reasons:

- we have to evaluate the radiance for each point of the scene in one direction (radiance is directional),
- to evaluate the radiance of a point, it is not necessary to consider all the points of the scene. Indeed, since the radiance is an integral, we can evaluate it by taking a certain number of samples of the variables of the integrand (Gauss or Monte Carlo methods),
- visibility between two points can be easily computed by ray tracing,
- extended form factors, in presence of non ideal specular surfaces, can be easily evaluated by tracing rays from point to point, while their evaluation is very difficult when using patches,
- several kinds of surface can be sampled into points: polygons, spheres, cylinders, cones, parametric surfaces etc.. Subdivision into patches can then be avoided, which allows the use of different kinds of geometric models.

3.1.2 The Discretization Method

In our present implementation, the scene is made up of a collection of convex quadrilaterals. The discretization process consists in recursively subdividing each quadrilateral in four subsurfaces. The recursion stops when the area of a subsurface is below a certain threshold fixed by the user. Once this subdivision has been accomplished, a sample point is placed at the center of each subsurface. With each sample point is associated the area δS of the surface containing it. These areas δS are used to evaluate the solid angles between two sample points. This process is illustrated by figure 2. We preferred to consider the centers of the subsurfaces as sample points, rather than the vertices, so as to avoid undesirable effects along the edges shared by two surfaces. Note that a better subdivision would be to add a second threshold for the differential solid angles between two samples as suggested in [15, 16].

3.2 The Discretized Light Energy Balance Equation

Since the scene is discretized into points, only the point-to-point light contributions have to be evaluated. In order to discretize equation (1), we exploit the fact that with each sample point of a surface S_k is associated a surface area δS_k. Equation (1) becomes then:

$$L_{ij} \approx h_{ij} \left[L_i^E + \sum_{k=1}^{N} R_{kij} G_{ki} L_{ki} \delta S_k \right] \qquad (2)$$

where N is the number of sample points in the environment. Using this point sampling and separating the diffuse and specular reflections, we obtain:

$$\begin{aligned} L_{ij} &= h_{ij}\left[L_i^E + L_i^d + L_{ij}^s\right] && (3)\\ L_i^d &= \sum_{k=1}^{N} d_i R_i^d G_{ki} L_{ki} \delta S_k = \sum_{k=1}^{N} \mathcal{D}(L_{ki}) && (4)\\ L_{ij}^s &= \sum_{k=1}^{n} s_i R_{kij}^s G_{ki} L_{ki} \delta S_k = \sum_{k=1}^{n} \mathcal{S}(L_{ki}), && (5) \end{aligned}$$

where L_i^D is the diffuse component of the reflected light, L_i^S the specular component and n is the number of sample points included in the *specular reflection cone* (or specular transmission cone) which bounds the specular component R^s of R. The angle of such cones depends of the roughness of the materials. The more important the roughness, the larger the angle. Such cones are defined by an axis which is the perfect reflection (or transmission) direction, and by an angle which depends on the physical properties of the materials (see figure 3). In our case, the reflection cone bounds the Beckman function D, since this latter models the roughness. In case of perfectly specular materials, the reflection cone is reduced to the perfect reflection direction. As no sample point will lie along this direction, we select the sample point which is closest to this direction. Note that the operators $\mathcal{D}$ and $\mathcal{S}$ are the global diffuse and global specular operators, respectively.

Since transmission is treated exactly analogously to reflection, we will omit it for now.

The diffuse component is evaluated by adding the contribution of all points in the scene to a given point, while the specular component takes into account only the points whose contribution will be significant, i.e. the points included in the reflection cone.

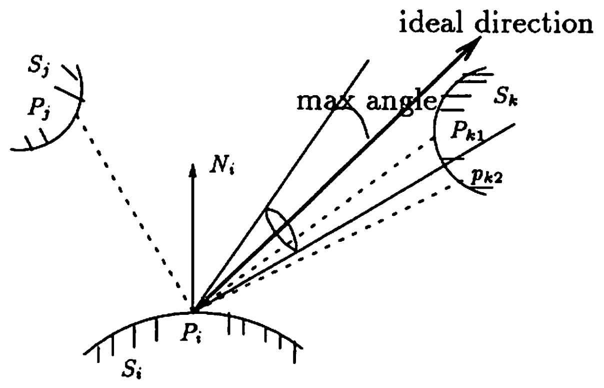


Figure 3: the reflection cone

3.3 First Pass

3.3.1 Extended Form Factors

Since a reflected light can be separated into diffuse and specular components [5], the two-pass method consists of two main steps: the first one is a view independent computation of the *Global Diffuse Radiance* while the second evaluates the view dependent *Global Specular Component*. Let us rewrite equation (1):

$$L_{ij} = h_{ij}\left[L_i^E + \sum_{k=1}^{N}(d_i R_i^d + s_i R_{kij}^s) G_{ki} L_{ki} \delta S_k\right]$$

$$= h_{ij}\left[L_i^E + \sum_{k=1}^{N} d_i R_i^d G_{ki} L_{ki} \delta S_k + \sum_{k=1}^{n} s_i R_{kij}^s G_{ki} L_{ki} \delta S_k\right]$$

$$= h_{ij}\left[L_i^E + \sum_{k=1}^{N} \mathcal{D}(L_{ki}) + \sum_{k=1}^{n} \mathcal{S}(L_{ki})\right]$$

The global diffuse radiance L_i^d of a point is then given by:

$$L_i^d = L_i^E + \sum_{k=1}^{N} d_i R_i^d G_{ki} L_{ki} \delta S_k = L_i^E + \sum_{k=1}^{N} \mathcal{D}(L_{ki}). \quad (6)$$

The expression of L_i^d, as function of L_j^d is then:

$$L_i^d = L_i^E + \sum_{j=1}^{N}(\mathcal{D} + \mathcal{D}(\sum_{k=1}^{n} \mathcal{S}) + \mathcal{D}(\sum_{l=1}^{n} \mathcal{S}(\sum_{m=1}^{n} \mathcal{S})) + \ldots) L_j^d.$$

The term $[\mathcal{D} + \mathcal{D}(\sum_k \mathcal{S}) + \mathcal{D}(\sum_l \mathcal{S}(\sum_m \mathcal{S})) + \ldots]$ is called *Extended Form Factor* and is named EFF_{ji} by analogy with radiosity [28, 6].
It is the proportion of diffuse light emitted by the surface S_j that contributes to the global diffuse radiance of the point P_i (see figure 4). Using these terms we obtain a system of N equations of N unknowns:

$$L_i^d = L_i^E + \sum_j \left[EFF_{ji} * L_j^d\right] \quad (7)$$

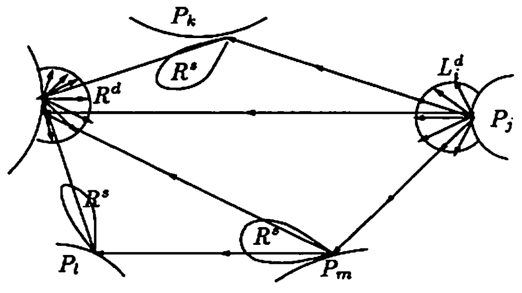


Figure 4: Extended form factor FFE_{ji}

This system can be solved (for each wavelength) by using the same methods as for radiosity (complete or iterative solution [9]).

3.3.2 Algorithm

The following algorithm computes the contribution of one given point to all its environment. This algorithm uses a shooting process since it allows a progressive solution.

```
/* statements */

/* matrix of spectral extended form factors */
spectrum $EFF_\lambda[N][N]$
/* $N$ is the number of sample points */

/* spectrum initialized to 1.0 */
spectrum spec1 = {1.0, ..., 1.0}

/* computes the j^th column of the extended form
factors matrix */
EvaluateEFF(j)

/* j:  emitting point */
{
    for all points $P_i$ {
        /* geometrical term */
        compute $FF_{ji} = h_{ji} G_{ji} \delta S_j$
        /* for each spectrum sample */
        $EFF_\lambda[j][i] = EFF_\lambda[j][i] + FF_{ji} * d_i R_{\lambda i}^d$
        if ($s_i \neq 0$) {  /* specular surface */
            GlobSpecOp(j, j, i, spec1, 1)
        }
    }
}
```

The procedure EvaluateEFF evaluates the direct contribution of the global diffuse radiance of P_j to the global diffuse radiance of all the points of the scene. All points being illuminated by P_j and belonging to a non perfectly diffuse surface emit the specularly reflected light to the environment. This is made possible thanks to the following procedure GlobSpecOp.

```
/* Global Specular Operator*/
GlobSpecOp(j, l, i, $\Delta EFF_\lambda$, lg)

/* j:  emitting point */
/* i:  last point met */
/* l:  previous point in light path */
/* $\Delta EFF_\lambda$:  cumulated contribution */
/* of $P_j$ to $P_i$ via $P_l$ */
/* lg:  length of path from $P_j$ to $P_i$ */

{
/* light is emitted by $P_l$ */
/* and reflected by $P_i$ towards points $P_k$*/

    /* geometrical term */
    compute $FF_{li} = G_{li}\delta S_l$
    for all points $P_k \in$ reflection cone {
        /* geometrical term */
        compute $FF_{ik} = h_{ik}G_{ik}\delta S_i$
        /* for each wavelength */
        $\Delta EFF_\lambda = \Delta EFF_\lambda * FF_{li} * s_i R^S_{\lambda lik}$
        $EFF_\lambda[j][k] = EFF_\lambda[j][k]$
            $+ \Delta EFF_\lambda * FF_{ik} * d_k R^d_{\lambda k}$
        if ((lg < lgmax)
            and($\Delta EFF_\lambda > EFFMIN_\lambda$)
            and($s_k \neq 0$)) {
            GlobSpecOp(j, i, k, $\Delta EFF_\lambda$,lg+1)
        }
    }
}
```

3.4 Second Pass

In the second pass (which is view dependent) the global specular operator is evaluated by means of a distributed ray tracing [10]. Note that this step does not entail the shooting of rays towards light sources (shadow rays), since the global diffuse component of the sample points (of the scene) is already available. Moreover, the intersection between a shot ray and the scene results in a point which may not be a sample point. However, the global diffuse component of the radiance at this intersection point can be interpolated by using the diffuse spectral radiances of the four closest sample points computed at the first pass as done in [2, 18].

4 Visibility Calculation

This section shows how the visibility between two sample points is performed. To evaluate the visibility between a pair of sample points P_i and P_j, a ray is cast from P_i toward P_j. If this ray hits one object before reaching P_j, then the two sample points do not see each other. This process entails a large amount of ray-object intersections and needs to be accelerated. To this end, two data structures are used: spatial subdivision and visibility graph.

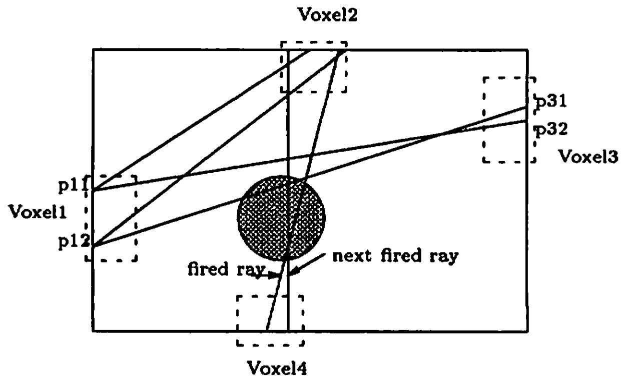

Voxel2
p31
p32
Voxel3
p11
Voxel1
p12
fired ray
next fired ray
Voxel4
Voxel1 and Voxel2 are fully visible
Voxel2 and Voxel4 are fully hidden
Voxel1 and Voxel4 are partially hidden:
the subgraph (p11,p31),(p11,p32),
(p12,p31),(p12,p32)
is stored

Figure 5: Voxel-to-voxel visibility information

4.1 Spatial Subdivision

The parallelepipedic bounding volume of the scene is recursively subdivided by planes aligned with the coordinate system axes. Each slicing plane subdivides a space into two equal sized subspaces. This subdivision results in a set of unequal sized subspaces which, from now, are called voxels. With each voxel is associated a list of sample points located in this voxel. This recursive subdivision stops either when the number of objects (which are convex quadrilaterals in our current implementation) intersecting the current voxel is below a certain threshold, or when the maximum level of subdivision is reached.
To apply Amantides's traversal algorithm [1], a spatial index SI is used. This spatial index is a $3D$ grid, whose each element $SI[i,j,k]$ is a pointer to a voxel (see [4] for more details).
In contrast to a uniform grid, our spatial subdivision into unequal sized voxels allow to reduce the amount of memory needed to store the visibility graph described hereafter.
Note that this spatial subdivision is also used in the second pass to calculate the global specular component by ray tracing.

4.2 The Visibility Graph

4.2.1 The Graph

Since the scene is sampled into points, it seems worthwhile to build a visibility graph giving a boolean visibility information for all pairs of points, instead of building a valuated graph whose complexity is $O(N^3)$ for each wavelength as done by Buckalew [7].

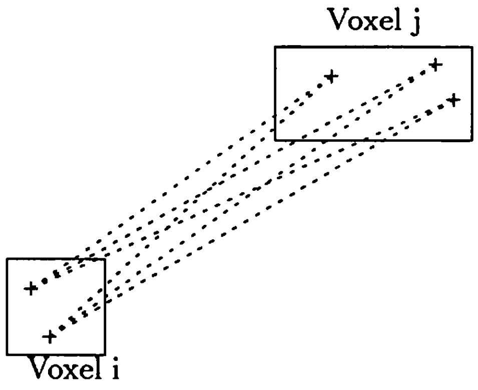


Figure 6: Visibility between two voxels

This visibility graph is very attractive but requires a large memory. To cope with this problem of memory size, we propose the following visibility strategy. Since the scene is subdivided into voxels, we build a visibility graph giving a visibility information between each pair of voxels instead of sample points. The visibility test checks how each voxel is visible from the other voxels. This test fires rays between the two voxels. Each ray corresponds to a pair of sample points belonging to different voxels as shown in figure 6.
The data structure used to store the visibility graph is :

```
Nedges = N(N+1)/2;
/*where N is the number of voxels*/
typedef struct /*edge data structure*/
{
boolean visibility;
object *buf-int;
booleanmatrix *voxel-graph;
}Edge;

/*visibility graph data structure*/
Edge Visibility-Graph[1..Nedges];
```

The nodes of the visibility graph are voxels, while its edges are elements of the linear array *Visibility-Graph[]*. The field voxel-graph is a pointer to a boolean matrix storing the visibility information between the points of the two voxels of an edge.
Let i and j be the two voxels of edge k. Three cases can be considered:

1. all points of voxel i are visible from all points of j.
```
Visibility-Graph[k].visibility = true;
Visibility-Graph[k].buf-int = null;
Visibility-Graph[k].voxel-graph = null;
```
2. the sample points of i do not see those of j.
```
Visibility-Graph[k].visibility = false;
Visibility-Graph[k].buf-int = null;
Visibility-Graph[k].voxel-graph = null;
```
3. the sample points of i see only a part of those of j (figure 5). In such a case we decide to store the visibility information between all pairs of points included in the two relevant voxels as well as the pointer to one object lying between these two voxels.
```
Visibility-Graph[k].visibility = false;
Visibility-Graph[k].buf-int =
    pointer-object;
Visibility-Graph[k].voxel-graph =
    pointer-matrix;
```

Note that the field *buf-int* of the data structure *Edge* plays an important role. Indeed, during the visibility test between two voxels, as soon as a fired ray intersects objects lying between the two voxels, only the pointer to the closest object is stored in the field *buf-int* of *Edge*. Due to the spatial coherence, the next fired ray has a great probability to intersect the same closest object between the two voxels. Consequently, this next fired ray will be checked for intersection with only this closest object, the pointer of which is already in *buf-int*, which saves a significant amount of computation. This approach seems to be a simplified version of the light buffer method [13].
In most scenes we have treated in our experiments, this strategy appeared rather efficient but we must keep in mind that the real memory complexity of this graph is always $O(N^2)$.

4.2.2 Using the Visibility to Improve the Scene Discretization

During the evaluation of the visibility graph, the distance between each pair of points is computed. If the distance between two points is small compared to their associated surface area δS (important solid angle), the corresponding surface elements are locally subdivided. We obtain then new sample points with smaller associated surface areas. Thereby, the solid angles between these two points become smaller, which makes the computation more accurate.

4.2.3 Other Advantages

As said above, the visibility graph contains only purely geometric information, independent of the photometric properties of the objects. This allows to modify these properties (reflectance, transmittance, roughness, self-emittance...) while keeping the same visibility graph. Indeed, to obtain new values of radiance, only a graph traversal is needed. Moreover, when a few objects are moved, only a small part of the visibility graph has to be modified. This graph might be updated with a rapid incremental method. This method is currently under investigation.

5 Results

We express the cost of the evaluation of the extended form factors matrix in term of numbers of calls to the visibility

number of voxels	102
number of empty voxels	5
number of pairs of points	16076090
number of pairs of visible voxels	100
number of pairs of hidden voxels	1009
number of pairs of partially hidden voxels	4042
number of pairs of points stored	5 889 008
graph computation time	39mn 49sec.

Table 1: Cost of the visibility graph

	With Graph	Without Graph
Time (seconds)	48mn 33sec	3h 24mn 12sec
Number of visibility computations	8038045	42 433 940

Table 2: Computation of the matrix of extended form factors

function h_{ij}. The cost of our method is given when the visibility graph is used, and when it is not. The results obtained are in favor of the use of this graph.
Our test scene is made up of 134 polygons. The materials of the objects of the scene are gold, sand, concrete, brown stone, blue and green enamel. All these materials are perfectly diffuse except gold (leg of the table) and silver (mirror) whose parameters are: $s = 0.9$, $d = 0.1$ and the roughness coefficient $m = 0.3$ (Beckmann term) for gold, and $s = 0.9$, $d = 0.1$ and $m = 0.05$ for silver. The scene includes two primary light sources which are normalized D6500 white sources. The sampling of the scene results in 4010 points, 196 of which lie on specular surfaces. To emphasize the influence of specular materials on the global diffuse radiance, our test scene was processed according to three different ways, giving the three following images:

image 1 (figure 7): all the materials are assumed to be perfectly diffuse,

image 2 (figure 8): scene containing diffuse and specular materials, image resulting from the first pass,

image 3 (figure 9): scene containing diffuse and specular materials, final image obtained after the two passes.

Note that the light spots near the leg of the table are the global diffuse component due to the specular properties of this golden leg. The spot light near the door is due to specular reflection on the silver mirror.
The amount of memory required to store (table 1) the visibility graph has been drastically reduced thanks to our non uniform spatial subdivision, whereas this amount is very important for a spatial subdivision into a regular $3D$ grid.
Among the 8.038.045 visibility calculations to be performed, only 1.394.017 of them are actually made. Indeed, the visibility computations corresponding to the following cases are avoided:

- sample points of the same surface;
- the angle formed by the ray direction and the normal at a sample point is greater than 90 deg;
- the buffer *buf-int* is used to evaluate the visibility function.

6 Comparisons with Other Algorithms

In contrast to our model, the global illumination algorithms described in [29, 28, 18] use an empirical reflection model, a trichromatic approximation, and an ideal specular reflection. Even though the algorithm in [27] uses a new physics-based reflection model, it is limited to ideal reflection. As for the multi-pass ones [26, 8], they seem more suited for rendering caustics but are very time expensive compared to the two-pass methods.

7 Conclusion

Unlike most of the models already implemented, our illumination model accounts for a physics-based reflection model, spectra instead of a trichromatic approximation, the spectral reflectance and transmittance of materials as well as color science. In our implementation, the scene is sampled into points instead of small patches. To prove that point sampling is correct, we have generated one image with that kind of sampling. It has been compared with the image of the same scene generated by a technique based on a discretization into patches. The visual results seem very similar. This point discretization offers the advantage of evaluating, very easily, the extended form factors when no assumption is made on the specular behavior of materials.
Even though the visibility graph requires an important memory size, it significantly reduces the synthesis time, and in addition, it is well suited to an adaptive point discretization that improves the precision of the solid angle calculations, which avoids thus all artifacts.

References

[1] J. Amanatides and A. Woo. A fast voxel traversal algorithm for ray tracing. In *EUROGRAPHICS'87 Conference Proceedings*, pages 3–10, August 1987.

[2] J. Arvo. Backward ray-tracing. In *Course of the SIGGRAPH'86. Developments in Ray-Tracing*, 1986.

[3] M. Born and E. Wolf. *Principles of optics*. Pergamon press, 1970.

[4] K. Bouatouch, M.O Madani, T. Priol, and B. Arnaldi. A new algorithm of space tracing using a CSG model. In *EUROGRAPHICS'87 Conference Proceedings*, pages 65–78, August 1987.

[5] C. Bouville and K. Bouatouch. A unified approach to global illumination models. In *PIXIM'89 Conference*, pages 250–263, September 1989.

[6] C. Bouville, K. Bouatouch, P. Tellier, and X. Pueyo. Theoretical analysis of global illumination models. In *Photorealism in Computer Graphics*, pages 57–71, EurographicSeminars, Springer-Verlag, 1992.

[7] C. Buckalew and D. Fussel. Illumination networks: fast realistic rendering with general reflectance functions. *Computer Graphics*, 23(3):89–98, July 1989.

[8] S. E. Chen, H. E. Rushmeier, G. Miller, and D. Turner. A progressive multi-pass method for global illumination. *Computer Graphics*, 25(4):165–174, August 1991.

[9] Michael F. Cohen, Shenchang E. Chen, John R. Wallace, and Donald P. Greenberg. A progessive refinement approach to fast radiosity image generation. *Computer Graphics*, 22(4):75–84, August 1988.

[10] R. L. Cook, T. Porter, and L. Carpenter. Distributed ray tracing. *Computer Graphics*, 18(3):165–174, July 1984.

[11] R.L. Cook and K.E. Torrance. A reflectance model for computer graphics. *ACM transactions on graphics*, 1(1):7–24, January 1982.

[12] M. Garcia, M. Perraudeau, and P. Chauvel. Experimental measurements of reflectance and transmittance with a spectrophotometer. February 1992. CSTB, private communication.

[13] E.A. Haines and D.P. Greenberg. The light buffer: a shadow testing accelerator. *IEEE Computer Graphics and Applications*, 6(9):6–16, September 1986.

[14] A. Roy Hall and Donald P. Greenberg. A testbed for realistic image synthesis. *IEEE Computer Graphics and Applications*, 3(8):10–20, November 1983.

[15] P. Hanrahan and D. Salzman. A rapid hierarchical radiosity algorithm for unoccluded environments. In *Photorealism in Computer Graphics*, pages 151–169, EurographicSeminars, Springer-Verlag, 1992.

[16] P. Hanrahan, D. Salzman, and L. Aupperle. A rapid hierarchical radiosity algorithm. *Computer Graphics*, 25(4):197–206, August 1991.

[17] X. D. He, K. E. Torrance, F. X. Sillon, and D. P. Greenberg. A comprehensive physical model for light reflection. *Computer Graphics*, 25(4):175–186, August 1991.

[18] P.S. Heckbert. Adaptative radiosity textures for bidirectional ray tracing. *Computer Graphics*, 24(4):145–154, August 1990.

[19] D. S. Immel, M. F. Cohen, and D. P. Greenberg. A radiosity method for non-diffused environments. *Computer Graphics*, 20(4):133–142, July 1986.

[20] Gary W. Meyer. Wavelength selection for synthetic image generation. *Computer Vision, Graphics, and Image Processing*, 41:57–79, 1988.

[21] S.P. Mudur and S.N. Pattanaik. Multidimensional illumination functions for visualization of complex 3d environment. *Visualization and Computer Animation*, 1(2):49–58, 1990.

[22] P. Poulin and A. Fournier. A model for anisotropic reflection. *Computer Graphics*, 24(4):273–282, August 1990.

[23] H. Rushmeier and K. Torrance. Extending the radiosity method to include reflecting and translucent materials. *ACM Transaction on Graphics*, 9(1):1–27, January 1990.

[24] B. Le Saec and C. Schlick. A progressive ray-tracing-based radiosity with general reflectance functions. In *PhotoRealism in Computer Graphics*, pages 101–113, EurographicSeminars, Springer-Verlag, 1992.

[25] M. Shao, Q. Peng, and Y. Liang. A new radiosity approach by procedural refinements for realistic image synthesis. *Computer Graphics*, 22(4):93–101, August 1988.

[26] P. Shirley. A ray tracing method for illumination calculation in diffuse-specular scenes. In *Graphis Interface Conference Proceedings*, pages 205–212, May 1990.

[27] F. Sillion, J. R. Arvo, S. H. Westin, and D. P. Greenberg. A global illumination solution for general reflectance distributions. *Computer Graphics*, 25(4):187–196, August 1991.

[28] F. Sillion and C. Puech. A general two-pass method integrating specular and diffuse reflection. *Computer Graphics*, 23(3):335–344, July 1989.

[29] J. R. Wallace, M. F. Cohen, and D. P. Greenberg. A two-pass solution to the rendering equation: a synthesis of ray tracing and radiosity methods. *Computer Graphics*, 21(4):311–320, August 1987.

[30] J. R. Wallace, K. A. Elmquist, and E. A. Haines. A ray tracing algorithm for progressive radiosity. *Computer Graphics*, 23(3):315–324, July 1989.

[31] R.C. Weast, D.R Lide, M.J. Astle, and W.H. Beyer. *CRC Handbook of Chemistry and Physics.* CRC Press, Inc. Boca Raton, Florida, 1989-1990.

[32] G. Wyszecky and W. S. Stiles. *Color Science, Concepts and Methods, Quantitative Datas and Formulas.* J. Willey and sons, 1982. 2nd Edition.

APPENDIX

Computing the Fresnel factor

We have implemented two methods to efficiently evaluate $F(\lambda, \theta)$.

First method

In several books [32, 3, 31], we can find, for several materials, Fresnel factor curves $F(\lambda, 0)$ for normal incidence, as well as the refraction index $\tilde{n}$ for the wavelength $\tilde{\lambda} = 589$ (Sodium D lines) which corresponds to the center of the visible bandwith. Given these data, $F(\lambda, \theta)$ can be approximated [11], for each wavelength, by:

$$F(\lambda_i, \theta) = F(\lambda_i, 0) + \left(F(\lambda_i, \frac{\pi}{2}) - F(\lambda_i, 0)\right) \frac{F(\tilde{\lambda}, \theta) - F(\tilde{\lambda}, 0)}{F(\tilde{\lambda}, \frac{\pi}{2}) - F(\tilde{\lambda}, 0)},$$

where $F(\tilde{\lambda}, \theta)$ is given by the Fresnel formula for $\tilde{n}$.

Second method

In [31], for several materials, values of the refraction index are given for a certain number of wavelengths. In this case, $F(\lambda, \theta)$ can be exactly expressed with the Fresnel formula.

Storage of $F(\lambda, \theta)$

Knowing the expression of $F(\lambda, \theta)$, we can precompute it for each sample wavelength and for different values of θ (20 seem enough). These values allow to create a look-up table, from which any $F(\lambda, \theta)$ can be computed by a simple linear interpolation.

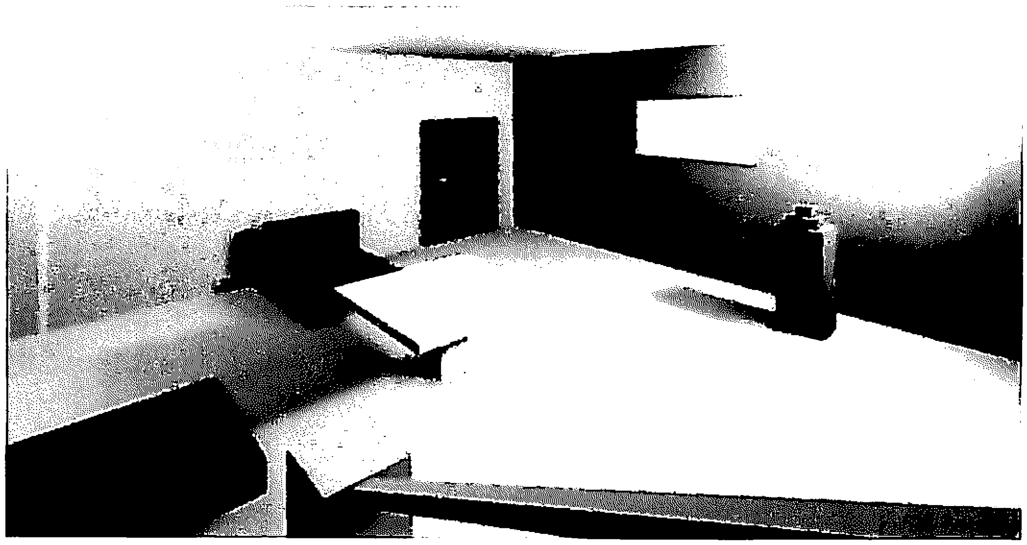


Figure 7: Image 1: result of the first pass; perfectly diffuse materials

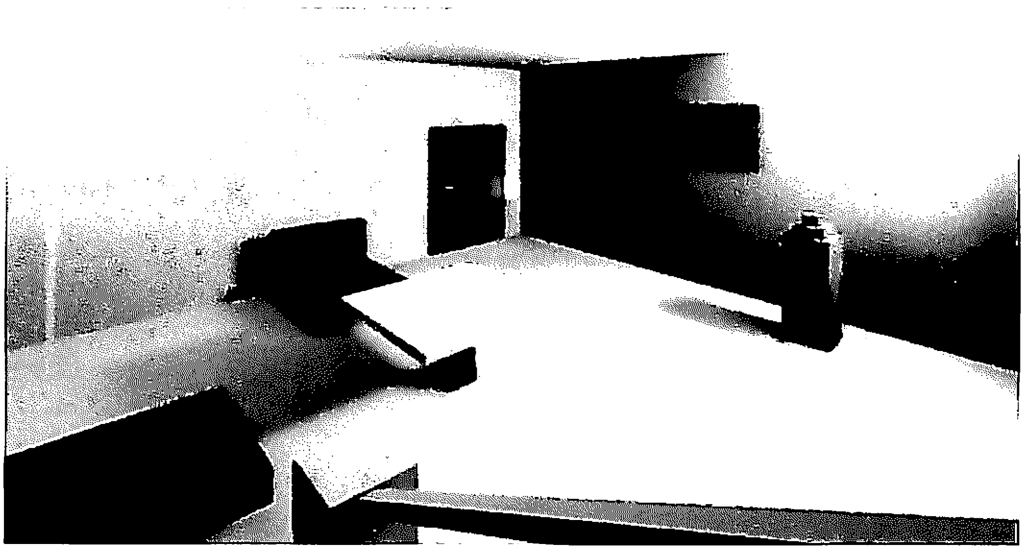


Figure 8: Image 2: result of the first pass; diffuse and specular materials

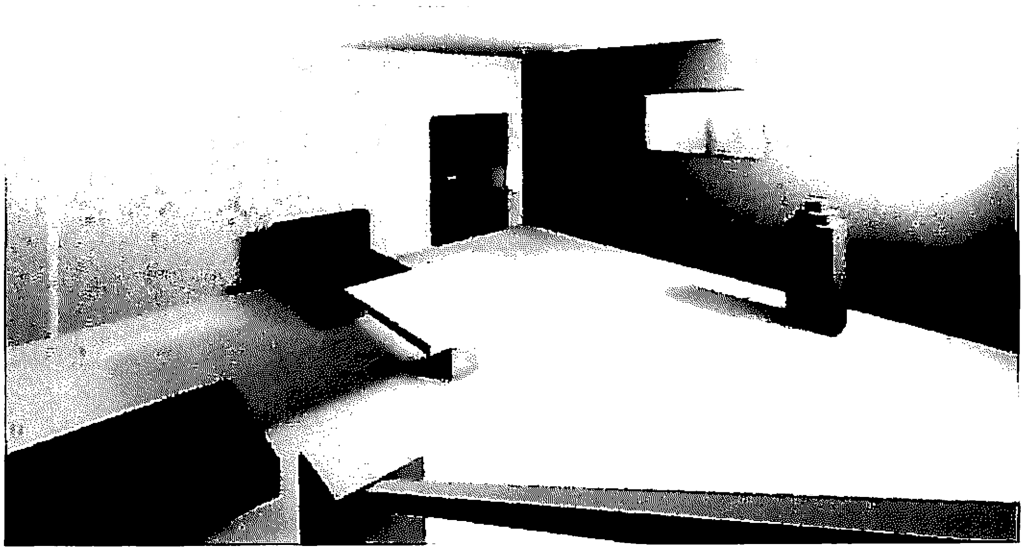


Figure 9: Image 3: result of the two passes

Author Index / Répertoire des auteurs